ABSTRACT

SHAH, ARPAN PRAMOD. Scalable authorization in role-based access control using negative permissions and remote authorization (Under the direction of Dr. Gregory T. Byrd).

Administration of access control is a major issue in large-scale computer systems. Many such computer systems proposed over recent years aim at reducing the effort required to govern access. Role-based access control (RBAC) systems are a huge benefit to this point. They reduce the tasks of an administrator or authorities when users take on different roles in an organization and need to be assigned different access rights or *privileges* based on these roles. RBAC is a very expressive and flexible access control mechanism that makes it possible to have security policies based on the principle of least privilege, static and dynamic separation of duties, conflicts between roles and permissions, and many more. This research proposes the use of *negative permissions* and *remote authorization* for improving the scalability of an RBAC implementation.

We discuss how negative permissions would fit in the proposed RBAC model. The thesis describes a mechanism to implement such an RBAC system utilizing negative authorizations. Our implementation is an extension of the Java 2 security architecture to support negative authorizations. We provide support for hierarchy of roles and de-confliction of positive and negative authorizations using the *most specific takes precedence* model. Future extensions to the model and optimizations to the implemented algorithm are proposed.

Another aspect of this thesis is the application of above RBAC model in a distributed environment utilizing a remote authorization management system. A remote authorization mechanism is appropriate in many client-server systems where there is control over the resources at an intermediate communication stack or a middleware component enforces the access rules. In our client-server architecture, an authoriza-

Report Documentation Page	*Form Approved OMB No. 0704-0188*

1. REPORT DATE **2003**	2. REPORT TYPE	3. DATES COVERED **00-00-2003 to 00-00-2003**

4. TITLE AND SUBTITLE **Scalable Authorization in Role-Based Access Control Using Negative Permissions and Remote Authorization**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **North Carolina State University,Department of Electrical and Computer Engineering,Raleigh,NC,27695**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**78**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

tion server uses an RBAC system to control access to resources under its domain, and the enforcement of access rules is provided by a *security overlay* on privileged resources.

We address how our negative permissions and remote authorization schemes augment RBAC scalability. We provide the requisite abstraction through UML and architecture diagrams for implementation in other languages and systems. A comparison of this work to other related research done in the RBAC domain is carried out, and future work in this area is discussed.

SCALABLE AUTHORIZATION IN ROLE-BASED ACCESS CONTROL USING NEGATIVE PERMISSIONS AND REMOTE AUTHORIZATION

by

Arpan P. Shah

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Masters of Science

Department of Electrical and Computer Engineering

Raleigh

2003

Approved By:

Dr. Gregory T. Byrd
Chair of Advisory Committee

Dr. Douglas S. Reeves

Dr. Peng Ning

To

Mom and Dad

Biography

Arpan Shah was born on the 10th of December, 1978 in the city of Mumbai, India. He completed his Bachelors of Engineering in the Electronics discipline from D.J. Sanghvi College of Engineering, Mumbai. Soon after, he worked for a year as a Software Engineer at one of India's most successful software services organization, Infosys Technologies Limited. He joined the Masters program in Computer Networking at North Carolina State University in the fall of 2001 and has been working under the guidance of Dr. Gregory Byrd as a Research Assistant in the field of access control for distributed environments since spring 2002. His research and development interests are network & information security, and wireless networking.

Acknowledgements

I would like to express my sincere appreciation to my advisor, Dr. Gregory Byrd, for his guidance and constant support. He has always kept confidence in me and his critical comments have significantly improved the content of this thesis and the maturity of research performed.

I am grateful to my committee members, Dr. Douglas Reeves and Dr. Peng Ning, for their valuable comments and suggestions.

Special thanks to my colleague Rong Wang from MCNC-RDI with whom i collaborated during the first semester in my research. My initial work with her laid the foundation for this thesis. Thanks to other colleagues in the Yalta project group: Praveen Cheruvu, T.J. Smith, Xiaoyong Wu, Krithiga Thangavelu, and Hongjie Xin for making this a rewarding team experience.

Thanks to Mr. Michael Warres from Sun Microsystems for being prompt and meticulous in our email correspondence.

I want to express my gratitude to my mom, Sharmistha Shah, for her unwavering faith in me, and my dad, Pramod Shah, for serving as a role-model in many of the roles i have performed to date.

This work was supported by the DARPA Information Technology Office (ITO), under contract F30602-00-C-0068.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Role-based access control (RBAC) [14] concepts have been utilized in computer systems for more than two decades by implementing group-based access control (Unix, Windows NT), and security policies based on the Chinese wall model for separation of duties [8]. RBAC simplifies the administration of access control by granting permissions to users on their roles and not on their identities. RBAC provides an abstraction layer between the users and their permissions in the form of roles. As permissions granted to a user are generally in direct correlation with the roles they perform, the role-to-permission mapping remains relatively static compared to a direct user-to-permission mapping. If the user changes roles, only the roles mapped to the user need to be changed and the permissions pertaining to new roles will be automatically applicable to this user. In some situations, this mechanism might become too restrictive and inflexible as some permissions might not directly correspond to their roles. This is solved traditionally by including the user identity as one of the roles of the user, for example, user accounts in Unix. For example, this would allow a particular co-founder of an organization to perform certain functions on the basis of this authority in the organization. In this approach, access is granted if the user is allowed to access the resource based on either the identity, or one or more roles.

A related issue in RBAC is that of *exceptions* in role-based access. One of the mechanisms to achieve these exceptions is to implement *negative authorizations* in the access control mechanism [42]. Negative authorizations can serve as over-riding cases to positive authorizations, or one can implement a de-confliction strategy to come out with a correct decision in case of simultaneous presence of positive and negative authorizations. A system that overrides positive authorizations by negative ones compromises the utilization of exceptions. A certain de-confliction strategy and implementation, which ensures that the security requirements of the system are met and yet is flexible for utilizing exception cases, serves a dual purpose.

1.1 Study

Over the last 10 years, role-based access control has emerged as the most favored system for access control. It simplifies access control administration to a large extent and minimizes errors due to human elements in configuration of the access control mechanism. However, the RBAC model has only recently matured enough to be considered for standardization [15]. We are yet to see widely accepted commercial implementations based on these RBAC standards proposed.

One of the most important benefits of RBAC is the simplification of administration associated with managing access in a moderate to large scale system. Exceptions to role based access control have not received significant attention in the research community. Exceptions tend to violate the basic principle of RBAC, that is, access based on roles. The research community considers constraints [2] as a suitable mechanism to achieve this objective. However, constraints should be used to provide pre-conditions to be satisfied before the access tuple checks in the system and are too complex for utilizing them for exception cases.

What needs to be pondered is that constraints serve as pre-conditions in the access check and that utilizing negative authorizations is the most natural mechanism to provide fine-tuned role-based access. Utilizing negative authorizations to always override positive authorizations is also not a flexible way of specifying policies. A more flexible approach is needed which takes into account the inheritance relationship between roles and gives appropriate preference to the positive or negative authorization based on these inheritance relationships.

A scalable system maintains the performance of a system in face of higher throughput that might be obtained by adding more resources. RBAC systems address the scalability issue of older access control models. A role-based system is inherently scalable in some aspects, due to a concise representation of access rights in the system. They cater to the inheritance of access rights found in real world scenarios and help remove redundancy in specification and issuance of access rights.

1.2 Implementation

This research has been complemented by a prototype implementation over the Java language security mechanism [20]. Java has a well-defined access control mechanism and security specification. Being a platform-independent language, an implementation in Java would be available to be used over all platforms over which Java is available. We utilize the user-centric access control mechanism [47] in Java and extend it to include negative authorizations.

We modified Java's security mechanism by changing a few authorization classes from the language. We also modified the *Overture* [48] toolkit, which utilizes classes that allows administrators to dynamically change the access control tuples at runtime.

We implemented this authorization in a distributed environment secured by a middleware framework [43]. The middleware framework is a *security overlay* that provides security services to applications. Our framework for authorization enables this middleware to consistently enforce the security functionalities expected from it. Administration of access control is also simplified in such a system. There are fewer points of contact for administration and enforcement of security functionalities.

We believe this remote authorization framework supports our case of augmenting negative authorizations to the RBAC model. In absence of negative authorizations, exceptions cannot be easily expressed. Many positive authorizations would have to be issued by the authority instead of a few negative authorizations to individuals and one positive authorization for the containing group. This would mean a large set of policy statements or tuples issued by the policy authority, causing a large number of entries in the security enforcement mechanism utilized by the authorization server. This would, in turn, mean a larger time to search for the correct tuple matching the request. For a large system, many such requests would be sent by clients to receive the policy decision and hence, an efficient search is required to save time. This search time, and policy issuance time and bandwidth can be saved by utilizing negative authorizations, as they reduce the number of policy tuples and their search time.

1.3 Outline

Chapter 2 provides the background information required for appreciation of this thesis. We provide a background of some security models and how RBAC can be accepted as a general model for access control. We discuss the various levels of RBAC, each higher level representing an increased sophistication of implementation. Section 2.2 describes the security mechanism in Java and how it can be used to implement RBAC. Also explained are the relevant Java APIs (Application Programmer's Interfaces) for utilizing the security mechanism implemented by Java.

Chapter 3 describes our implementation. Section 3.3 explains how we extended the Java API to include negative authorizations with no modification required to any existing application code. Section 3.4 explains the implementation of a remote authorization scheme to cater to access control requests. We show how this is possible in a manner that is transparent to applications. We demonstrate the flexibility of our mechanism by configuring the system to implement local policy enforcement or utilize the remote authorization server without any code level changes. We discuss the scalability of our remote authorization architecture and our policy distribution mechanism that guarantees confidentiality and data origin and content integrity. Section 3.5 presents the merits and limitations of this work.

Chapter 4 compares our work with prior work done in the area of negative authorizations and constraints. We contrast some earlier work done in remote authorization for role-based access control with our mechanism.

Chapter 5 concludes the thesis and suggests some directions for future work.

Chapter 2

Background

Increased dependence of humans on computer systems mean high damage if their vulnerability is exploited. This damage can range from passive threats, like eavesdropping on some confidential data, to active threats, like identity-theft, where a malicious user poses as an authorized user and then carries out transactions on that authorized user's account. As vulnerable these computer systems are, they have to be protected from being abused by unauthorized users. Various models are used to categorize these protection mechanisms and to help us develop an approach to prevent or counter-attack these threats.

Fundamentally, these protection mechanisms fall into three broad categories: Confidentiality, Integrity, and Availability. Maintaining confidentiality is to prevent disclosure of information to unauthorized people. Integrity is concerned with correct detection of the origin of information and the ability to guarantee that information being shared was not modified in transit. Availability pertains to the presence of information or a service whenever a legitimate user needs to access it. A security overlay is a layer of these security services built on top of lower level services.

Any model helps us formulate the rules that conforming implementations should provide. A model gives us an abstraction for further high-level development and leads to a robust implementation. A security model is such a specification that details how a particular system achieves the three fundamental security goals of confidentiality, integrity and availability. Many models in computer systems target a particular kind of security goal and provide a solution for that particular goal.

2.1 Access Control Models

Access control is a means to provide confidentiality in a system by granting or denying the right to access data or perform some action (method). Access control models aim to provide solutions to the confidentiality threats in a system. Traditionally, these access control models have been based either on a mapping of what resources users can access or on how flow of information between various objects in the system is controlled. These models, known as discretionary access control (DAC) and mandatory access control (MAC) respectively, are discussed in subsequent sections.

Resources governed by access control are termed *privileged resources* because accessing them needs some privileges on the part of users. In an object-oriented system like Java, resources are modeled as objects having some data and operations to be performed on this data. Privileged objects can only be accessed by users having certain privileges sufficient to access them. Each user has a set of *credentials*, which enables that user to be granted appropriate privileges. This requires that the requester first be identified and validated as a legitimate one*(Authentication)* and then the appropriate decision taken to grant or deny access*(Authorization)*. By performing authentication, we extract the credentials possessed by a user and try to prevent illegitimate use of resources by under-privileged users. These credentials are granted in the form of private and/or public credentials and are attached to a *principal.* A user can have more than one set of credentials for different contexts, and he can present one or more sets of these credentials representing one or more principals for accessing a resource. This collective set of principals along with the credentials is known as a *Subject* in literature. A subject can be considered to be either a user or a program running on behalf of the user. Since authorization decision is based on this set of principals and their credentials, we say that a subject is the unit of access control. To summarize, access control is a mechanism to prevent privileged operations on protected objects from unauthorized subjects.

Access control can also be used in an *audit trail* to provide accountability in the system. This is done by documenting the access of objects by a particular subject and recording what operations were performed.

Role-based access control is a recent addition to the traditional discretionary and mandatory access control models and has received significant attention in research. Next, we discuss these access control models individually.

2.1.1 Discretionary Access Control (DAC)

Discretionary access control [36] has traditionally been used by operating systems and relational database systems. In systems conforming to the DAC model, usually a system administrator sets the initial privileges of each user, based on the user's roles and needs and the policies of the organization. Each user, in turn, has the discretion to grant operations to other users on the objects that belong to this user. Access control is generally user-based, though one can have variations such as group-based access control provided in systems based on Unix and Windows NT operating systems. A DAC system is modeled in the form of an access matrix, which is a two-dimensional matrix with subjects presented as rows and objects as columns.

Authorization information can be placed with the subjects in a *capability-based* system or with the objects in an *access control list* model. An example of a capability-based system is Kerberos [29]. Capabilities take the form of tokens that are required to access a privileged resource and these tokens can be easily copied or delegated. Figure 2.1 shows an example of some capabilities that users might possess for accessing privileged resources. In order to be useful, capabilities must not be forgeable. Capabilities also offer a better review of access privileges of subjects than the access

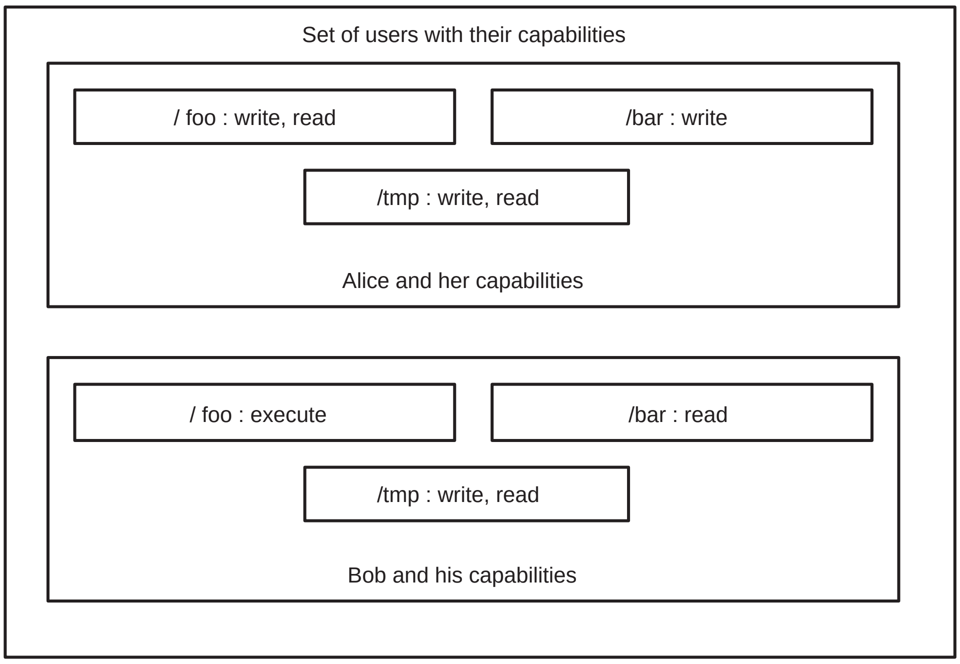
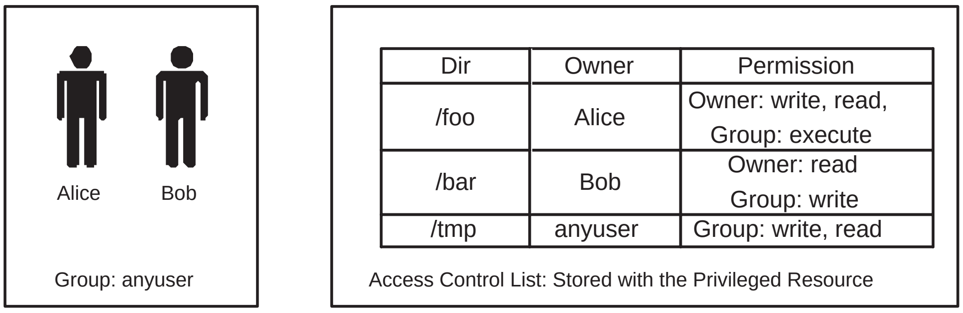


Figure 2.1: Capabilities

control list (ACL) model used in Unix based operating systems.

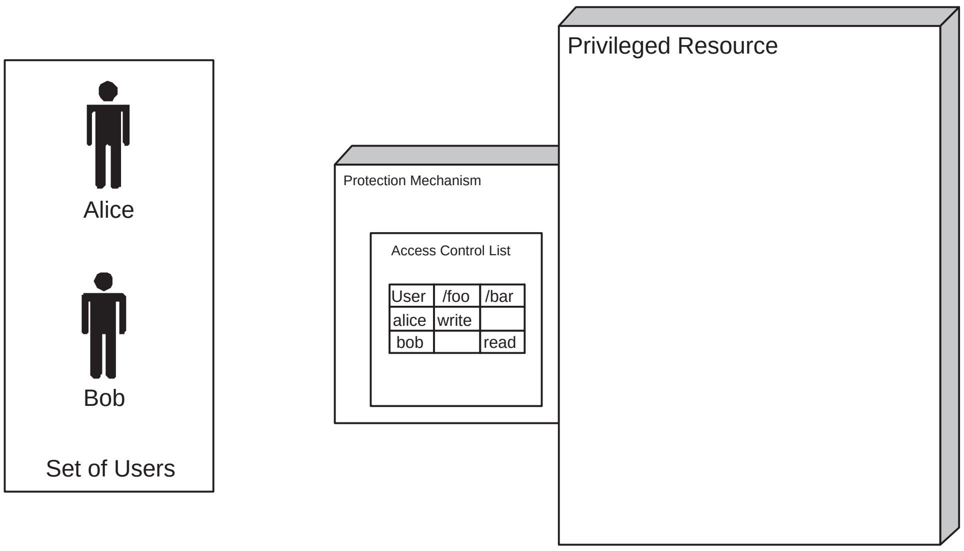
ACL is a list, typically present at each protected resource, that stores which subjects are allowed to access its information or perform an operation. Hence, ACLs offer a better review of per object access control rather than per subject access control. Figure 2.2 shows an example of an ACL present at a privileged resource for access control.

ACLs are more common in a closed system like an operating system, whereas capabilities are traditionally preferred in some distributed systems like an organization intranet. Figure 2.3 shows an example of an access control matrix where the subjects are shown as rows and objects as columns in a two-dimensional matrix. Access to privileged resources is restricted by a protection mechanism that bases its decision on a discretionary access control mechanism (which is an access control list in this case).



Dir	Owner	Permission
/foo	Alice	Owner: write, read, Group: execute
/bar	Bob	Owner: read Group: write
/tmp	anyuser	Group: write, read

Figure 2.2: Access control lists



User	/foo	/bar
alice	write	
bob		read

Figure 2.3: Discretionary access control

2.1.2 Mandatory Access Control (MAC)

Discretionary access controls are not sufficient to control flow of information from one object to another. They have no control on what information can be copied between objects [37]. This problem becomes more subtle when a program executing on behalf of one of the trusted users tries to maliciously leak information from a privileged resource accessible to the user. This kind of subverting of security is known in literature as the *Trojan Horse* attack. Mandatory access control is a mechanism to counter such attacks.

Mandatory access control is concerned with the control of flow of information from one object in a system to another. The notion of object in MAC encompasses the notion of object and subject used in DAC models. In this system, each object has a security classification known as a *label.* This label is a *security classification* for objects and *security clearance* for subjects. The label determines whether one object is able to access another object and what flow of information is allowed between these objects. Bell and LaPadula formalized this approach of mandatory access control [5]. The central idea in this approach is to utilize discretionary and mandatory access control to enforce these information flow policies. This acts as a two-tier security approach wherein subjects need to have discretionary rights over the objects in addition to mandatory rights. Subjects do not have any control over the mandatory rights. Rights are determined by their security label and the label of the object they are trying to access.

Figure 2.4 shows how mandatory access control techniques are adequate at enforcing flow policies for access control. Alice tries to execute a malicious program written by bob. The program executes with Alice's privileges and hence the security clearance of the program is *Secret.* The program does the intended work that Alice expects it to be done. However, along with that, it reads /foo directory files and writes them to /bar directory. In absence of MAC, Bob is allowed to read these /bar directory files, and so he will be able to read those files even though he does not have the direct

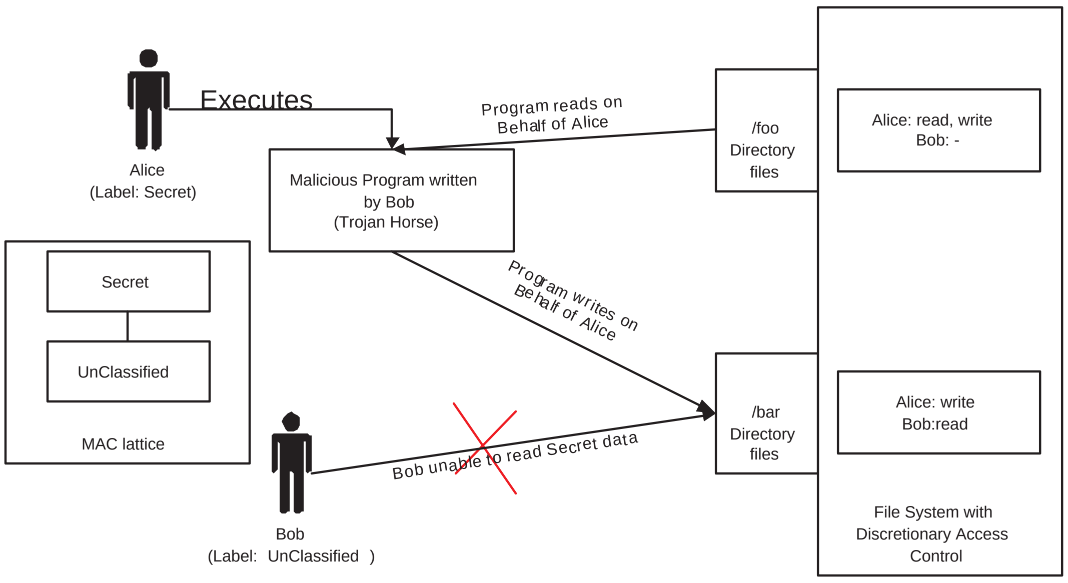

Executes
Alice
(Label: Secret)
Malicious Program written by Bob (Trojan Horse)
Program reads on Behalf of Alice
/foo Directory files
Alice: read, write
Bob: -
Secret
UnClassified
MAC lattice
Program writes on Behalf of Alice
/bar Directory files
Alice: write
Bob:read
Bob unable to read Secret data
Bob
(Label: UnClassified)
File System with Discretionary Access Control

Figure 2.4: Trojan Horse problem

permission to read the file in /foo directory. However, if a MAC system is used along with the DAC mechanism, Bob would not be able to read the file written by Alice's program, as it will be written with the security label *Secret*. Thus, problems due to such Trojan Horse programs are prevented by utilizing a MAC system.

MAC models are further specified into MAC models for confidentiality, integrity, or composite models, depending on whether the flow model intends to provide confidentiality, integrity, or both. In a MAC system, access is permitted to a subject only when the information flow is to an object whose label is dominated by the subject's label. Which label dominates another is governed by the *security policy* in effect which can cater to confidentiality, integrity or both. In the MAC confidentiality model for files, if a user's label dominates that of the file then the user will be able to read the file as information flows from the file to the user. A MAC model is implemented in many military environments and in other systems containing highly sensitive data.

2.1.3 Role-based Access Control (RBAC)

Unlike identity-based DAC models, access control in an RBAC system is based on one or more roles that a user performs. RBAC models provide a significant benefit to the administration of access control because users are granted access to resources based on particular roles they perform in the organization. Roles are assigned to users, and permissions are assigned to roles. Thus, roles can be viewed as providing an intermediate layer for access control that groups users dynamically according to their functions in the organization. The most important benefit of a role-based system is its ease of administration. Even if a user's role changes in the organization, the only change required for access control purposes is the allocation of the user to that new role. Permissions to the users are implicitly assigned. The newly assigned roles contain these permissions.

RBAC is a more general mechanism than DAC or MAC. RBAC can be configured to provide DAC or MAC support as it can embody a particular security policy for implementation [32]. To configure RBAC to a DAC implementation, a set of administrative roles is required. One administrative role is for the owner of a privileged resource. Other administrative roles are pertaining to users to whom the owner may grant privileges for the resource. Besides the administrative roles, one role is required per privileged resource for specifying what actions are permissible on the resource. Configuring RBAC to a MAC implementation requires that the security labels be identified as roles in the system and the information flow conditions regarding the label domination be specified as *constraints* in the model. Constraints act as preconditions before any resource access and enable us to specify more complex policies which cannot be implemented by DAC or MAC.

Figure 2.5 shows what happens when a user tries to access a privileged resource governed by an RBAC system. The first step is authentication of the user to determine identity of the user. Typically RBAC systems implement some form of a database which stores the roles that a user is in or asks the user to present security credentials

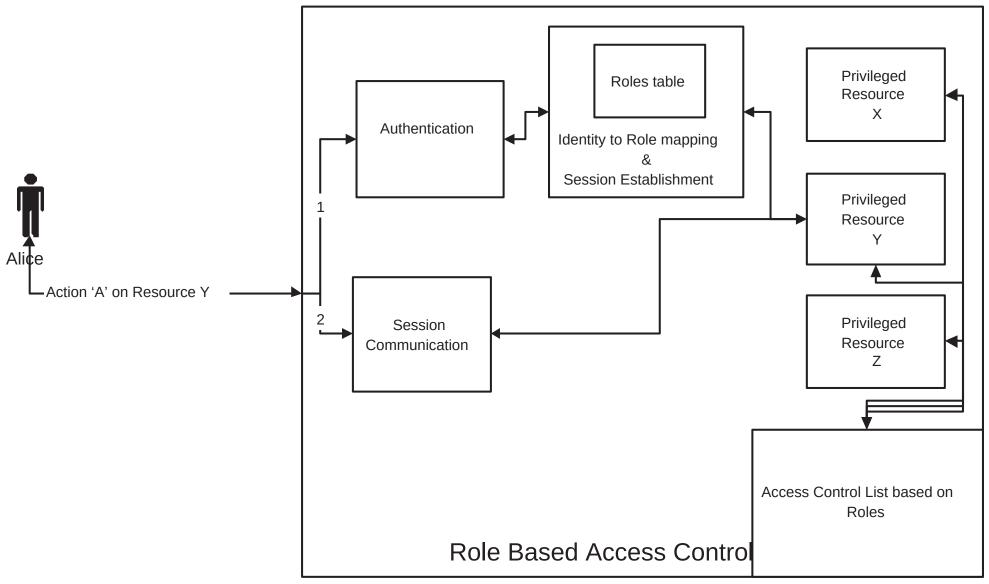


Figure 2.5: Role-based access control

to determine which roles the user wants to be activated for the current access. These roles remain activated for the session, depending on what session semantics the RBAC system employs. The database that stores the access privileges is not stored on the basis of user identity, but the roles that user is in. This mechanism allows ease in administration and a rich set of policies to be implemented via sophisticated RBAC techniques.

2.1.3.1 RBAC Levels

RBAC models have been qualified into four increasingly sophisticated levels which enable researchers and developers to compare their systems with this reference [38].

RBAC0 is the base model in this reference, which defines different entities in the overall model. As shown in Figure 2.6, RBAC0 consists of users, roles, permissions, and sessions. The permissions are always positive and can apply to single objects or many. Also present in the model are the user-role and role-permission assignment relations. Both the relations are many-to-many. This means that a user can belong

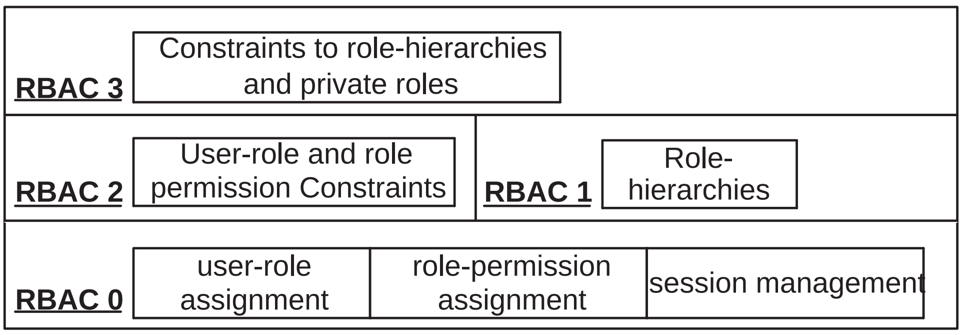

RBAC 3
Constraints to role-hierarchies and private roles
RBAC 2
User-role and role permission Constraints
RBAC 1
Role-hierarchies
RBAC 0
user-role assignment
role-permission assignment
session management

Figure 2.6: RBAC levels

to many roles and a role can have many users. Similarly, a role can have many permissions, and the same permission can be assigned to many roles. Sessions are established by users when they activate one or more roles to perform some privileged operations. Each session is associated with a single user. However, a single user might have multiple sessions open at the same time, depending on the security policy of the organization.

RBAC1 introduces the concept of role hierarchies to the RBAC0 model. They leverage a very important structural heritage of roles: inclusion of junior roles into senior roles. Junior roles are the ones that are less powerful than the senior roles. Senior roles inherit all the permissions of the junior roles, thereby removing redundancy in role-permission assignments and simplifying administration of access control rights. Further concepts such as depth of inheritance and private sub-hierarchies are also introduced, which help us map this model to the real world scenario.

RBAC2 introduces the concept of *constraints* to the RBAC0 model. Constraints are a mechanism that help an organization lay out a higher level policy that has to be honored before every access. Constraints can apply to user-role, role-permission assignments and other factors such as time criteria to be followed before every access. An important constraint used to prevent abuse of authority is the constraint on roles to be mutually exclusive. This is related to the principle of separation of duties [18]. A similar constraint on mutually exclusive permissions also supports this principle of separation of duties for permissions. Constraints act as prerequisites on roles and permissions that any subject has to pass in order to be granted the requested role / permission.

Access control constraints [2] can also be used to cater to *exception* cases. Constraints, however, are typically used as pre-conditions for access to be granted to any role and are implemented via a separate mechanism. Thus, constraints are not added to any access control rules or tuples. Instead, a separate table that has to be checked before each access. Utilizing constraints for access control purposes would require that these constraints be added to any tuple that has an exception case with it. Such constraints would require a complicated parser and are not implemented by any existing access control mechanisms to our knowledge.

RBAC3 subsumes the RBAC1 and RBAC2 approaches and also defines additional constraints such as those on role hierarchies and private roles. Private roles are roles that do not inherit the permissions assigned to their junior roles.

A parallel set of levels have been also introduced for administration of an RBAC model [14]. These models are simpler than the corresponding RBAC models and a lower level administrative model can be used to administer a higher RBAC model. These administrative models modify the user assignments, permission assignments, role hierarchy relations, and constraints. However, for a small organization, a single administrator can directly take care of access control using RBAC instead of using an administrative model for control.

In this era of Internet and enterprise applications, RBAC implementations would typically be applied along with other techniques such as trust management and digital rights management for a comprehensive protection mechanism [33]. A programming language such as Java or other platforms such as operating systems and relational databases would typically implement security primitives to utilize them.

2.2 Java Security Mechanism

As a programming language frequently used over distributed platforms like the Internet, one of the highest priorities of Java has been to provide strong security features. Java has strong security features to protect code and control what a code can access or execute via techniques such as byte code verification, strong type-checking, and embedded security checks in system classes [20]. The Java 2 architecture security [19] has evolved from the sandbox model, which either completely trusted the code (inside the sandbox) or distrusted it (outside the sandbox), to a more fine grained approach wherein custom security managers control what a particular code can access or execute. The latest Java standard edition development kit [47] (J2SE) includes Java Cryptography Extension (JCE), Java Secure Sockets Extension (JSSE), and Java Authentication and Authorization Services (JAAS). JCE is an extension of the Java Cryptography Architecture (JCA) aiming to provide more robust implementation of security protocols for supporting confidentiality and integrity. JSSE is a framework and an implementation supporting Secure Sockets Layer (SSL) and its IETF counterpart Transport Layer Security (TLS). SSL and TLS are the de facto standards on web security, providing point-to-point confidentiality and integrity [12]. JAAS is a framework for authentication and authorization useful in multi-user environments. These frameworks aim to provide security services with minimal security code embedded in the application.

2.2.1 Java Policy

In the default implementation provided by J2SE, a static system-wide policy object that is loaded at runtime from a *policy file* controls access to privileged resources like files, sockets, etc. The policy file contains all the permissions that are granted to specific locations from where the source is downloaded. This location is known as a codesource, and the Java run-time environment loads these permissions in the policy object. This is based on the premise that a user downloading code from different codesources would associate different privileges to the code depending on trust worthiness of the location.

An example of such a permission is given below.

```
grant codebase "http://url.com", {
   permission java.io.FilePermission "/foo/-", "read,write";
};
```

The above statement in the policy file would grant read and write permissions to any file in the /foo directory or any sub-directory, provided the code is downloaded from http://url.com.

In addition to such permissions, J2SE now supports granting permissions not only to codesources, but also to specific principals. These principal specific permissions can be granted access to resources based on the credentials presented during a session as mentioned in Section 2.1.3. This functionality is provided by the JAAS mechanism and is discussed in the next section.

2.2.2 Java Authentication and Authorization Services (JAAS)

For a multi-user environment like an enterprise application or a public Internet terminal, it becomes essential to base access control not on the location (codesource) from where the service was requested, but on who runs the code. This was the primary motivation that led to the development of JAAS. Now an integrated component of J2SE, JAAS enables authentication of users and making an authorization decision based on their credentials, thereby providing user-centric access control. User authentication in JAAS is performed in a pluggable fashion and hence application code for authentication operations are independent of the underlying authentication mechanism. JAAS effectively takes the Java 2 security architecture one step forward by providing user-centric access control as compared to the code-centric access control approach provided traditionally in Java.

Authentication can be done by developing modules for all authentication technologies such as Kerberos, username-password pair, etc. Authorization decision is based on the credentials that are collected from the user during authentication. These credentials take the form of different principals collectively represented by a subject which is explained in Section 2.1. JAAS conforms to the access control security model where authentication and authorization is based on subject having a set of principals having credentials.

Thus, Java now provides a unified mechanism for access control based on users, location of the code, and the context of the security manager.

A typical JAAS based permission grant in the J2SE policy file looks like this:

```
grant codeBase "foo.bar", principal foo.Principal "John" {
   permission java.io.FilePermission "/tmp/*", "read,write";
};
```

The above statement in the policy file would grant read and write permissions on any files residing directly inside the /tmp directory to 'bar' class file in 'foo' package running on behalf of a subject containing the principal 'John'. Here 'John' represents an instance of class 'Principal' in the 'foo' package.

2.2.3 Java Security Classes and API

Here we explain the Java security classes to help the reader better understand the security check performed by Java and the data structures these classes utilize to perform a security check. Java security mechanism is activated at run-time by installing a `SecurityManager` that acts as a gateway for all permission checking. It resolves the security checks to the current `AccessController` which has a snapshot of the current `AccessControlContext` in which they have to be performed. Without activating a `SecurityManager`, none of the access checks will be executed and all access will be granted in the system. Each AccessControlContext keeps a mapping of all the threads currently running within its context specified by CodeSource, Principal, and ClassLoader together grouped into a ProtectionDomain. A ClassLoader loads classes and makes them available to the Java run-time. When a ClassLoader loads a class, it associates this class with a Permission set granted to that class. These permissions are obtained from a Policy object that acts like a database of permissions. In the default Java implementation, this policy is loaded at run-time by reading a file into the Policy object. This file contains all permissions to be loaded into the system on startup for privileged resources. Each privileged resource typically has its own implementation of the Permission interface. For example, java.io.FilePermission is a Permission class used to protect file operations. Note that in all access checks, the security check is carried out against all the associated access control contexts of the executing thread. This means that when an application thread tries to access any resource, multiple checks will be performed with access control contexts varying from the application context to the Java system context. The final access check is with respect to the Java run-time, which is fully trusted.

There are several ways in which authorization checks are performed in Java, depending on the practical requirements of a system [4]. The authorization check models in J2SE version 1.4 are as follows:

1. Access based on the ProtectionDomain set of the current thread. This method is the most frequently used one and is called when a protected resource calls SecurityManager.checkPermission() for associating the ProtectionDomain associated with that resource by the AccessController of the current thread. For each access control context of the current thread, the functional stack is typically as follows:

```
SecurityManager.checkPermission(somePermission)
AccessController.checkPermission(somePermission)
//ProtectionDomain now bound to the thread
```

```
AccessControlContext.checkPermission(somePermission)
```

2. Access based on the `ProtectionDomain` of the `doPrivileged` caller. This method is typically used when a class is trusted to perform an action safely and securely. The access check in this mechanism pertains to `ProtectionDomains` only associated with the class that called `doPrivileged()` or below it in the stack. The functional stack is as follows:

```
AccessController.doPrivileged(somePermission)
AccessController.checkPermission(somePermission)
//ProtectionDomain now bound to the thread
AccessControlContext.checkPermission(somePermission)
```

3. Access based on the `ProtectionDomain` of another thread. This is typically used in a system when a proxy object has to perform the action requested by a user. For example, in a client-server system, the server would typically use this call by passing the context of the client. The server would call `AccessControlContext.checkPermission(somePermission)` with the AccessControlContext of the client in this case.

4. Access based on the `Subject` identity. This is the access mechanism that is user-centric and is used in applications that require access control based not only on the codesource, but also on who is performing the action. This is done by calling `Subject.doAs()` or `Subject.doAsPrivileged()` by passing the current subject, the `PrivilegedAction` to be performed, and the access control context if the call is `doAsPrivileged()`. `Subject.doAsPrivileged()` behaves similar to `Subject.doAs()` , except that instead of retrieving the current Thread's `AccessControlContext`, it uses the provided `AccessControlContext`. If the provided `AccessControlContext` is null, this method instantiates a new `AccessControlContext` with an empty collection of `ProtectionDomains`. This is used where a particular class is trusted to perform a dangerous action

safely [4]. One can think of this call as providing a similar functionality to *setuid* in Unix and *Impersonation* in Windows [11]. In Java, this would be useful in a client-server environment wherein the server would extract the client's subject and perform the operation by passing the client's `AccessControlContext`.

```
Subject.doAs(subject, privilegedAction)
privilegedAction.run()
SecurityManager.checkPermission(permission)
//Subject now bound to context of thread
AccessController.checkPermission()
AccessControlContext.checkPermission()
//Subject information taken into account
```

We utilized the access mechanism based on the `Subject` executing the method. This was because, we wanted to utilize the user-centric access control provided by Java to implement a role-based access control system utilizing negative authorizations.

2.3 Dynamic Policy Provider

The Java policy API provides a pluggable mechanism to build custom *policy providers*. While J2SE 1.4 does not provide dynamic granting of permissions at run-time, one can build on top of the APIs provided by Java to have such an implementation. One such implementation is the `DynamicPolicyProvider` class by the Overture [48] toolkit. The `DynamicPolicyProvider` extends the static policy in J2SE to include dynamic permissions for principals. `DynamicPolicyProvider` has multiple `SubPolicy` objects, depending on the `ClassLoader` that loads privileged classes. This implies that resources that base their access control on the same `ClassLoader` will have a specific `SubPolicy`. Each `SubPolicy` stores the permissions using a `DomainPermission` that contains a `Principal` set and a `PermissionCollection` which stores the permissions that are granted to these principals. The `SubPolicy` keeps a `Hashtable` of such `DomainPermissions` with the corresponding ProtectionDomain as the keys. When these DomainPermissions are initialized, they inherit all the permissions appropriate to their ProtectionDomain from the SubPolicy's basePolicy Object. This basePolicy Object is similar to what one would get on loading a static policy file in J2SE 1.4. When dynamic permissions are granted, the granted permissions are added to a Grant object based on the ClassLoader of the Permission class.

DynamicPolicyProvider takes a lazy-update approach, in which the grants are stored according to Set of Principals. Whenever a user tries to access a privileged resource it provides its Principals, which are a part of the running thread. The AccessController of the thread then queries the current SubPolicy for the ProtectionDomain that these Principals lie in and creates a new one if not already existing. If a ProtectionDomain already existed, then the SubPolicy checks in the corresponding DomainPermissions object to see if the PermissionCollection present *implies* the resource access permission. Whenever a new permission is entered into Policy, it is linked to all the matching ProtectionDomains.

The above mechanism thus provides a caching mechanism for the policy checks, thereby decreasing the access check time. DynamicPolicyProvider also provides for a review of Permissions associated with the ProtectionDomains via the getPermissions() method.

2.3.1 Access Check Algorithm

Following is the access check algorithm that Java security mechanism follows for Subjects that try to access privileged resources.

- Application tries to access a privileged resource.
- The resource has implemented a privileged operation and has a security check embedded via the call Subject.doAs (Subject subject, PrivilegedExceptionAction action).

- This invokes the corresponding `AccessController.doPrivileged (PrivilegedAction action, AccessControlContext context)`.
- Since action is a `PrivilegedAction`, this calls the current `AccessController.checkPermission(Permission perm, Object context)`.
- This calls the current thread's `AccessControlContext.checkPermission (Permission perm)`.
- This calls the appropriate `ProtectionDomain.implies(Permission permission)`. If the permissions inside this `ProtectionDomain` are static, the permissions reside in the `ProtectionDomain` itself and an exception is thrown if access is not granted. If the permissions are not static, then the current `Policy` is consulted.
- If `DynamicPolicyProvider` is installed as the current `Policy`, this invokes `DynamicPolicyProvider.implies(ProtectionDomain domain, Permission permission)`.
- This invokes the `SubPolicy.implies(ProtectionDomain pd, Permission p)`.
- This invokes the appropriate `DomainPermissions.implies (Permission p)`. `DomainPermissions` maps the current `Permissions` available to the specified `Principals`, and hence this method checks the `PermissionCollection` object.
- This checks the `PermissionCollection.implies(Permission p)` method.
- For all matching `Permission` objects inside this `PermissionCollection` object, each checks whether the `Permission` is granted or not by the `Permission.implies(Permission p)` method. This last call returns with either true or false, indicating whether the `Subject` was privileged to performed

the `PrivilegedAction`. If a Permission is not allowed, the invoked method returns with an `Exception` to indicate that the action is not allowed.

We shall see in Section 3.3, how we modified the `DynamicPolicyProvider` to include a parallel set of data structures for an access control list that stores negative authorizations in addition to positive authorizations. The UML diagram of the `DynamicPolicyProvider` will also indicate the removal of the caching mechanism, as this could mean more processing to update or invalidate the whole cache whenever a new authorization is added to `DynamicPolicyProvider`.

2.4 Distributed Computing Technology in Java

The above section introduced some security classes to provide an overview of the Java security mechanism and how access checks are made against the policy. This study would help the reader better understand our negative authorizations scheme. Next, we introduce two technologies in Java that make distributed computing easier. These technologies were extensively utilized by our remote authorization scheme.

2.4.1 RMI

The Java Remote Method Invocation (RMI) [44] utilizes a client-server model of distributed computing and provides for communication between remote programs written in the Java programming language. RMI allows an object running in one Java Virtual Machine (JVM) to invoke methods on an object running in another JVM. Services provide their interfaces to clients, and clients call the methods defined in the interfaces. To make this network communication transparent to the underlying service and client implementation, RMI uses the *Client-Dispatcher-Server* design pattern where the dispatcher is actually a *Distributed-dispatcher* pattern. Figure 2.7 shows the communication taking place in an RMI based client-server interaction. The client-side and server-side proxy classes perform the task of *marshalling* and *unmarshalling*

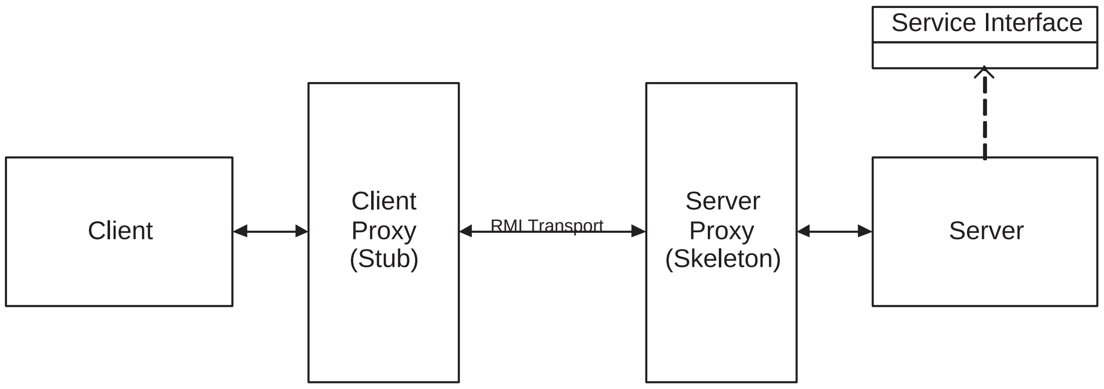


Figure 2.7: RMI

objects that are sent between the virtual machines for communication. Marshalling and unmarshalling are required for data to be sent over the wire for communication. Marshalling is storing the state of an object to a suitable form that can be sent over the wire and unmarshalled at the other side to get the same state of the object that was sent.

2.4.2 Jini

Jini [46] is a programming model that provides the various services and API required for building and deploying distributed systems. Jini utilizes RMI mechanisms and builds upon them to provide lookup services, discovery protocol, leases, event mechanisms, and transaction support amongst other services to manage distributed systems. While RMI's client interacts with the client side service proxy (Stub), a Jini client need not necessarily be a stub and can be the service object itself. Besides this, Jini provides a much more expressive mechanism to register and discover services; it provides a discovery service to enable clients to find services without knowing the location *a priori*.

We utilized a distributed system that used Jini and RMI for distributed computing. This distributed system was protected by the security services provided by an

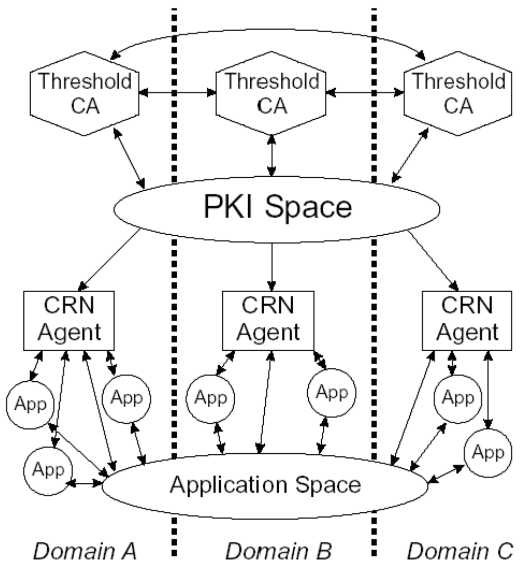


Figure 2.8: Yalta infrastructure

overlay service, called Yalta [9]. The next section discusses this infrastructure.

2.5 Implementation Infrastructure (Yalta)

Yalta is a security overlay service build for dynamic distributed environments such as coalitions between countries for war-time data sharing. For the prototype implementation, Yalta overlay is built on top of a distributed shared memory architecture provided by JavaSpaces [45]. We built on top of the available confidentiality and integrity services provided by Yalta to include access control mechanism into the overlay. This overlay provides a secure, scalable and reliable platform for information sharing by providing confidentiality, integrity, authentication and access control for distributed services. Figure 2.8 illustrates the infrastructure provided by the overlay. The trust management services are handled by Certificate Authorities (CA), and Certificate Revocation Notification (CRN) hierarchies and are intrusion tolerant and scalable. The CA signs the certificate request from the PKI space, which is the coalition infrastructure JavaSpace used to get certificate signing requests by the coalition members. These signed certificates are distributed through the PKI space, and the distribution hierarchy of CRN agents interact with application space to provide certificate revocation notifications through Certificate Revocation Lists (CRLs). The application space is also a JavaSpace with the Yalta security overlay mechanism.

The infrastructure is based on Java distributed computing technology such as RMI and Jini. Confidentiality services are provided by unicast security mechanisms utilizing RMI communication over SSL protocol, and multicast security is provided by secure group communication mechanism using Secure Spread [3]. Thus, confidentiality, integrity and authentication are provided by the trust management infrastructure, RMI over SSL, and secure spread. Access control was another important requirement for this overlay. Requirements for such a mechanism are explained in Section 3.1.

Chapter 3

Implementation

3.1 Requirements for Access Control In Yalta

Access control provided by JAAS is not sufficient for open, distributed environments. Such environments require access control privileges to change dynamically when the trust between participating members or the number of participating members changes. This requires use of dynamic permissions provided by Overture [48]. It also requires the use of sophisticated techniques like role-based access control. For the prototype implementation, we have not focused on how a role is derived. This could be done by mechanisms such as a role attribute assigned in identity certificates of users or in separate attribute certificates distributed through another hierarchy. We have embedded the role information inside the identity certificate for simplicity. This allows us to focus on providing negative authorizations as a feature in an RBAC system. We modified Overture's `DynamicPolicyProvider` class to include revocation of earlier granted permissions and to support negative authorizations.

The application space, as shown in Figure 2.8, is also built on the JavaSpace shared memory architecture. This means that the security overlay has control over resources in the application space and can embed security mechanisms in the application space. Thus, we could utilize access checking locally in the application space or remotely by implementing policy services. A local access check mechanism would require the access control policy to be enforced locally at each application space. Hence, we would require either a mechanism to either propagate this policy over all application spaces or an administrator per application space who would enforce the policies to the space. However, a more convenient and scalable mechanism would be to implement a policy service that is queried whenever any application tries to perform an operation on the application space. This requires implementing a policy service and a mechanism to query this service by the application space. This policy service should be reliable, scalable, and fault tolerant, in accordance with the other components in the overlay. Thus, policy enforcement must be carried out only in these policy service components and only authentication would be performed locally by the application space. This mechanism has several benefits:

- No complex policy distribution hierarchy is required.
- Accidental or intentional abuse of access control policy by local authorities can be prevented or minimized.
- A remote access control decision service can make the architecture scalable for large number of privileged resources.

To summarize, the requirements for an access control system built for Yalta are:

- Dynamic granting and revocation of rights.
- A denial mechanism for flexible and concise policy representation as discussed in Chapter 1.
- A remote access control decision service that is secure, reliable, scalable and fault tolerant.

- An administrative interface that can be applied at the policy enforcement point irrespective of whether the enforcement is at remote authorization server or at the local resources.

We describe the implementation of a system that meets these requirements in Sections 3.3 and 3.4.

3.2 Extending the RBAC model

Access control policies are best expressed by retaining the natural way people express the access control required for a system, and this provides easier management of the system. These policies can naturally be expressed in terms of positive and negative authorizations, based on what the organization wants to let users access and what it wants to prevent them from accessing. Negative authorization policies can also be used to temporarily remove access rights from subjects if the need rises, although this same need can be addressed by revoking an earlier granted right. Many operating systems have traditionally supported negative authorizations such as Windows NT/2000, with the first matching access control element deciding whether access is granted or denied [11].

Negative authorizations can be used for providing exceptions in access control. In a certain organization, assume that all employees except contract professionals should have access to the company's new project proposals. Without negative authorizations, these can only be achieved by specifying positive authorization to all roles which are in the same branch as the contract professional roles and no authorization for the contract professional roles. However, by utilizing negative authorizations, this can be implemented in a concise manner by specifying positive authorizations to the most junior role and negative rights only to the contract professional role. This also means that any new role that will be added to the system will automatically inherit the most junior role's positive authorization, and would be beneficial in cases where this is the expected behavior.

Negative authorizations can also be very helpful in case of incremental updates to the policy. For example, suppose that an organization traditionally allows every employee, permanent or contract based, to access its new project proposals. Suppose that the organization has in all about hundred roles for permanent employees and about five roles for contract employees. Now suppose a new policy is passed that only permanent employees should be allowed to access new proposal data. In absence of negative authorizations, the earlier positive authorization will have to be retracted and many new positive permissions will have to be instead added, depending on the hierarchy in the organization. With negative permissions, only five negative permissions need to be added to the existing policy with minimal disruption to the authorized users due to the change of policy. Consider the unstable state when authorization statements are cryptographically signed and take significant time to be generated because of these security operations. Negative authorizations are especially useful in these cases as they would require minimal new authorizations to be specified.

One more way to have the same effect as negative authorizations would be to utilize a constraints mechanism with each access permission element. However, a complex parser-generator is required to support access control constraints at a single permission level.

To support our view for negative authorizations, let us consider an assumed organization Acme, Inc. If only positive authorizations were supported, then the introduction of a single contract role would be accompanied by the change in all earlier rows or tuples of the access control elements. There would be need for as many statements to be issued by the administrator to grant positive access to each permanent role in the organization.

```
grant principal FooPrincipal "bar" {
   permission FooPermission "/foo/bar/-", "read,write";
};
```

If constraints were supported at an individual tuple level, this would require a complex parser-generator for interpreting, creating and enforcing policy tuples. Such a tuple would look like:

```
grant principal FooPrincipal "*" except "contract" {
   permission FooPermission "/foo/bar/-", "read,write";
};
```

However, if negative authorizations were supported, only a single additional tuple would need to be added to the existing policy in the organization. Such a tuple would look like:

```
deny principal FooPrincipal "contract" {
   permission FooPermission "/foo/bar/-", "read,write";
};
```

Note that in addition to these tuples, the constraints and grant-only mechanisms would also require the retracting of earlier grants present in the system.

Thus, we propose that addition of negative authorizations to the RBAC model enhances the expressiveness of the model and provides better control over resource access. In Sections 3.3 and 3.4 we describe our implementation of these negative permissions for a role-based access control mechanism in Java and a remote authorization mechanism suitable in environments where authorities have access rights over resources and want to enforce the access control policy of resources in a scalable manner.

3.3 Negative Permissions

3.3.1 Approach

One of the main implementation considerations was to adhere to Java's security mechanism as much as possible. So we required a solution which would be conformant to Java's security mechanism and be efficient. Following were the options for implementing negative authorizations.

1. Implement additional methods to resources, one per original method, to imply a negative right or a deny.
2. Implement inverse permission classes to that of the existing ones to imply negative rights.
3. Maintain two data structures, one for positive authorizations and one for negative authorizations, and de-conflict the two lists to produce the correct result on access request.

The first two options were not chosen because Java's already existing security mechanism would be hampered by these approaches. For example, a Socket Permission would also have to be modified to have either ∼accept, ∼connect, ∼listen, and ∼resolve in `SocketPermission` class or we would need a class ∼`SocketPermission` that has the original functions signifying negative permissions. As this would require modifications to many existing Java classes, we chose the option of maintaining two data structures for authorization policies, one with positive authorizations and one with negative authorizations. This would minimally impact the existing security mechanism for Java internal permission classes and have only minor or no changes to the existing APIs for security.

We chose to utilize the `DynamicPolicyProvider` class for extending the policy mechanism, since it provides a dynamic way to grant permissions and has all the data structures in place to maintain the list of permissions granted. Our negative permissions requires a list similar to the one that `DynamicPolicyProvider` already manages. One of the main implications of this negative list would be to de-conflict negative permissions with the positive permissions and come up with an assured answer that is fail-safe.

3.3.2 Permission De-confliction

Permission de-confliction is a crucial part of an access control methodology supporting negative permissions. We adopt a fine-grained control in the de-confliction algorithm with respect to which permission overrides the other. Many contemporary systems use negative permissions as a means to over-ride positive permissions to treat the exception cases. This would work fine for a small enterprise where the number of role-hierarchies are limited. This mechanism is inflexible for large businesses having a much larger role hierarchy.

Our de-confliction algorithm is based on the premise that the policy administrator knows the role hierarchy when the policy is loaded into the system. In case of grant-only access control systems supporting a hierarchy, the hierarchy has to be known as the permission propagation depends on this hierarchy and hence the premise appears to be a fair one. For our implementation, we chose this hierarchy to be given in terms of a file read at the startup by the policy provider. The `PolicyEngine` maps these principal types and their hierarchy in a `Hashtable` to give the numerical value that resembles their hierarchy. For e.g., for a file containing

```
org.mcnc.anr.yalta.jaar.OrganizationPrincipal
org.mcnc.anr.yalta.jaar.GroupPrincipal
org.mcnc.anr.yalta.jaar.IdentityPrincipal
```

`OrganizationPrincipal` has value 1 (2^0), `GroupPrincipal` has value 2 (2^1), and `IdentityPrincipal` has value 4 (2^2). This numerical value of a principal type effectively represents the priority of that principal. This is in conformance to our earlier discussion that more specific principals would be given higher priority compared to generic principals that are higher in the hierarchy. Table 3.1 gives a precise idea of how these numerical values map for each and every principal type present or absent in a request for accessing a resource.

Higher the numerical value of the principal set, higher is the priority for deciding whether access should be given to the requester. Thus, for a particular resource

Principal	*Meaning*	*Value*
anr,john	john, only when he represents the anr group	3
- ,john	john, irrespective of any of the group he is in	2
anr, -	anr group member, irrespective of the identity	1
- , -	any member of any group in the organization	0

Table 3.1: Hierarchical principal-value table

method, if deny (*, John) and grant (anr, *) are present then access would be denied to John for that method of the privileged resource. This is in accordance to the theory that authorizations for most specific roles should take priority over other applicable authorizations.

3.3.3 Class diagram of Dynamic Policy Provider

We produce here the class diagram of the `DynamicPolicyProvider` that we modified to implement negative authorizations. This model helps us provide an abstraction for implementation in other languages and systems. Along with the class diagram of `PolicyEngine`, this would better illustrate the algorithm used in de-confliction of negative and positive authorizations.

As shown in Figure 3.1, `DynamicPolicyProvider` is the public implementation class that a client application needs to communication with. Since this class implements Java's policy interface, it is consistent with the existing policy calls and add news dynamic policy calls to the system. An administrator needs to call `grant(...)`, `deny(...)`, `removeGrant(...)`, and `removeDeny(...)` functions in order to add policy statements in the system and `getNegativePermissions(...)`, and `getPositivePermissions(...)` to request what permissions are granted to `ProtectionDomains`. The Java `SecurityManager` calls the `implies(...)` method for checking whether a permission is allowed to a domain. The `addPolicy(...)` method adds a policy specific to a particular `ClassLoader`. There is also a provision for

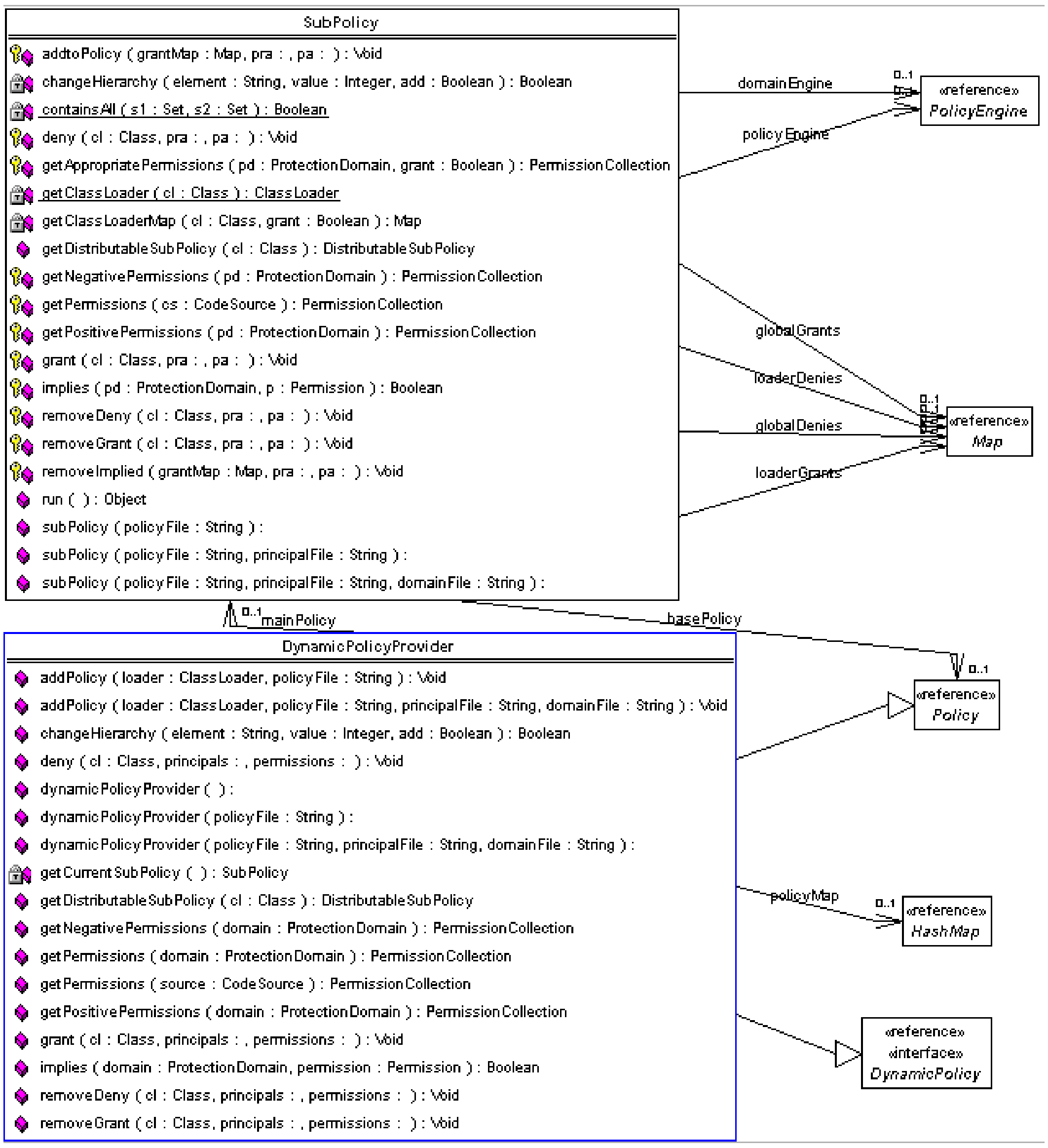

SubPolicy
addtoPolicy (grantMap : Map, pra : , pa :) : Void
changeHierarchy (element : String, value : Integer, add : Boolean) : Boolean
containsAll (s1 : Set, s2 : Set) : Boolean
deny (cl : Class, pra : , pa :) : Void
getAppropriatePermissions (pd : ProtectionDomain, grant : Boolean) : PermissionCollection
getClassLoader (cl : Class) : ClassLoader
getClassLoaderMap (cl : Class, grant : Boolean) : Map
getDistributableSubPolicy (cl : Class) : DistributableSubPolicy
getNegativePermissions (pd : ProtectionDomain) : PermissionCollection
getPermissions (cs : CodeSource) : PermissionCollection
getPositivePermissions (pd : ProtectionDomain) : PermissionCollection
grant (cl : Class, pra : , pa :) : Void
implies (pd : ProtectionDomain, p : Permission) : Boolean
removeDeny (cl : Class, pra : , pa :) : Void
removeGrant (cl : Class, pra : , pa :) : Void
removeImplied (grantMap : Map, pra : , pa :) : Void
run ( ) : Object
subPolicy (policyFile : String) :
subPolicy (policyFile : String, principalFile : String) :
subPolicy (policyFile : String, principalFile : String, domainFile : String) :
domainEngine
policyEngine
0..1
0..1
«reference»
PolicyEngine
globalGrants
loaderDenies
globalDenies
loaderGrants
0..1
«reference»
Map
0..1
mainPolicy
basePolicy
0..1
DynamicPolicyProvider
addPolicy (loader : ClassLoader, policyFile : String) : Void
addPolicy (loader : ClassLoader, policyFile : String, principalFile : String, domainFile : String) : Void
changeHierarchy (element : String, value : Integer, add : Boolean) : Boolean
deny (cl : Class, principals : , permissions :) : Void
dynamicPolicyProvider ( ) :
dynamicPolicyProvider (policyFile : String) :
dynamicPolicyProvider (policyFile : String, principalFile : String, domainFile : String) :
getCurrentSubPolicy ( ) : SubPolicy
getDistributableSubPolicy (cl : Class) : DistributableSubPolicy
getNegativePermissions (domain : ProtectionDomain) : PermissionCollection
getPermissions (domain : ProtectionDomain) : PermissionCollection
getPermissions (source : CodeSource) : PermissionCollection
getPositivePermissions (domain : ProtectionDomain) : PermissionCollection
grant (cl : Class, principals : , permissions :) : Void
implies (domain : ProtectionDomain, permission : Permission) : Boolean
removeDeny (cl : Class, principals : , permissions :) : Void
removeGrant (cl : Class, principals : , permissions :) : Void
«reference»
Policy
policyMap
0..1
«reference»
HashMap
«reference»
«interface»
DynamicPolicy

Figure 3.1: Class diagram of `DynamicPolicyProvider`

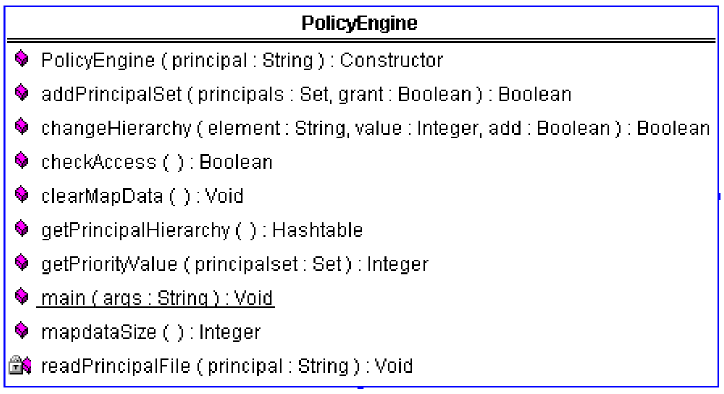

PolicyEngine
PolicyEngine (principal : String) : Constructor
addPrincipalSet (principals : Set, grant : Boolean) : Boolean
changeHierarchy (element : String, value : Integer, add : Boolean) : Boolean
checkAccess ( ) : Boolean
clearMapData ( ) : Void
getPrincipalHierarchy ( ) : Hashtable
getPriorityValue (principalset : Set) : Integer
main (args : String) : Void
mapdataSize ( ) : Integer
readPrincipalFile (principal : String) : Void

Figure 3.2: Class diagram of `PolicyEngine`

changing the hierarchy of principals so that the user does not need to re-configure every policy statement when a principal hierarchy changes. For our prototype implementation, this `changeHierarchy(...)` method is not implemented and can be subclassed in a system which needs to add its own method for implementing functions on change of hierarchy, such as backing up current policy object, or making appropriate modifications to the affected policy statements.

As discussed in Section 2.3, `DynamicPolicyProvider` acts as a consolidator of Java's static policy object and `SubPolicy` objects which actually store the dynamic policy information. For each `addPolicy(...)` function called with a different `ClassLoader` as the parameter, there is a `SubPolicy` object associated with it. This allows the Java virtual machine to have different policies for each `ClassLoader`.

The class responsible for confliction resolution for positive and negative permissions is the `PolicyEngine` class as shown in Figure 3.2. There is a `PolicyEngine` for each `ClassLoader` in a `SubPolicy` in the system. Each `PolicyEngine` contains information about domains that have the same principal hierarchy. Each `PolicyEngine` object stores in a `HashMap` the principals that a particular requester presents to access a resource. It also has, in a `Hashtable`, the priority values of each principal type in the system. Whenever an access request is to be resolved, the `PolicyEngine` calls the `checkAccess(...)` method to make the decision. Internally,the `checkAccess(...)`

method searches the `SubPolicy` object to find each applicable positive and negative permission and stores them sorted in ascending order in a `Hashtable` with priority value as the key and a boolean value indicating true or false as the value. Once all the applicable permission values from both the lists are put into this `Hashtable`, the last key's value gives the result. Optimizations possible to this algorithm are discussed in Chapter 5.

The above implementation is compatible with the earlier grant-only mechanism, except the `getPermissions(...)` method has to be deprecated. `getPermissions(...)` method returns a `PermissionCollection` object that can contain a set of positive-only or negative-only permissions. Returning only one of these permissions would no longer carry the correct semantic sense. Hence, `getPositivePermissions(...)` and `getNegativePermissions(...)` functions were added to the `DynamicPolicyProvider` to get the set of positive and negative permissions respectively for a particular `ProtectionDomain`. We could have implemented a backwards compatible solution by returning only the positive permissions in the `getPermissions(...)` method, but chose not to do so for correct semantics of our implementation.

An analysis of this implementation is presented in Section 3.5.

3.3.4 Local Enforcement Framework

This `PolicyEngine` and `DynamicPolicyProvider` were used along with the JAAS mechanism to provide local enforcement of access control policies at privileged resources. As shown in Figure 3.3, privileged resources enforce the access control mechanism in their JVM, which enables them to locally check whether access is allowed or not. This is in accordance to many existing discretionary access control frameworks where access control information resides with the privileged resources themselves. We implemented an `AccessManager` component to configure and startup the access control functionalities for the local enforcement mechanism. Later we shall describe how this same component was extended and utilized to provide a flexible enforcement

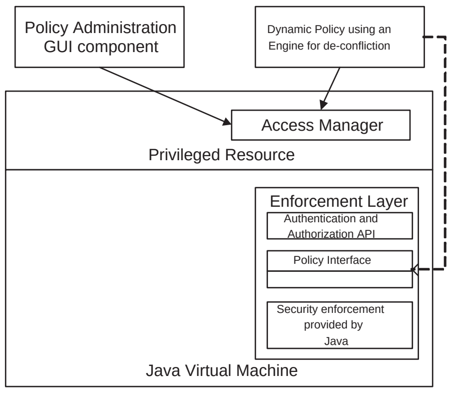

Policy Administration GUI component
Dynamic Policy using an Engine for de-confliction
Access Manager
Privileged Resource
Enforcement Layer
Authentication and Authorization API
Policy Interface
Security enforcement provided by Java
Java Virtual Machine

Figure 3.3: Local enforcement framework

mechanism which can be configured for remote authorization or local authorization. This `AccessManager` component has all the requisite configuration for both local and remote authorization. Thus, privileged resources do not have to be modified in any way for enforcing the policy locally or remotely.

3.4 Remote Authorization

As discussed in Section 3.1, implementing access control at a remote server in the presence of overlay services would enable ease in administration and prevent accidental or minimize intentional abuse by authorities at local resources. This means an extension to the current Java security mechanism where a client JVM makes remote calls to a `PolicyService`.

The primary consideration was to provide this extension as a plug-in to the existing JAAS mechanism. This plug-in would require only minor configuration changes to existing applications and would be calling the same API for access control as done by the JAAS mechanism. An application calls `Subject.doAs(...)` or its variant `Subject.doAsPrivileged(...)` to perform an action on a privileged resource. There were two options considered for this extension:

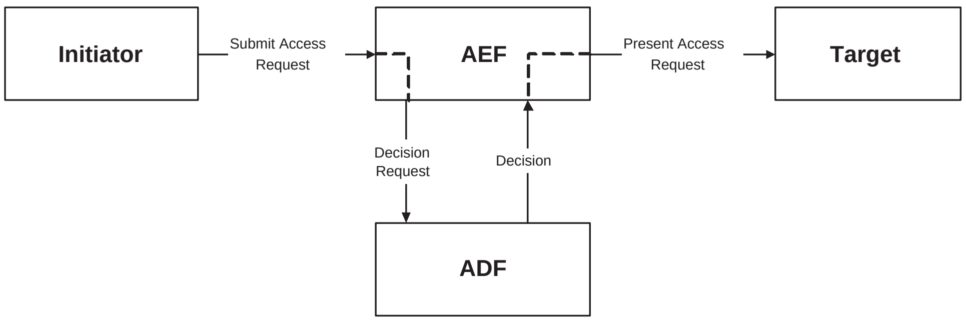


Figure 3.4: ISO-10181-3 access control framework

1. `Subject.doAs(...)` calls `SecurityManager.checkRemotePermission(...)` instead of `SecurityManager.checkPermission(...)`. This implies that this function would have to be written in the `SecurityManager` in Java.

2. `Subject.doAs(...)` calls `SecurityManager.checkPermission(...)` but when `Permission's implies(...)` method is called, then this permission makes a remote call to obtain the authorization decision. This means that we create a permission `PrivilegedPermission` that applications must extend for using remote authorization.

3.4.1 Design

Our remote authorization infrastructure is based on the ISO-10181-3 Access Control Framework [1], shown in Figure 3.4.

A user, who is the Initiator in this case, tries to perform an operation on a privileged resource by providing some information to the Access Control Enforcement Functions (AEF). This information is known as the Access Control Decision Information (ADI) in the ISO framework. The AEF makes a call to the the Authorization Decision Function (ADF) passing the ADI and additional context information needed for the access decision. The ADF then makes the authorization decision and sends this decision back to the AEF. If the operation is permitted, the AEF then performs

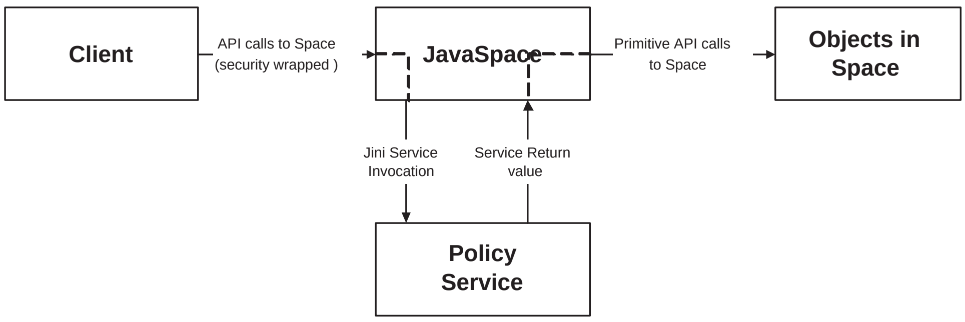


Figure 3.5: Remote authorization design

the operation on behalf of the initiator.

Figure 3.5 shows an example of how we adhered to the ISO framework for client communication with the application space. The initiators, in our case, are the clients that interact with the application space. Application space acts as the AEF and submits the decision request to the `PolicyService` which acts as the ADF. To submit this request, the application space first authenticates the client and then performs the requisite operations to extract the subject from the certificates (ADI) presented by the client. The `PolicyService` checks the enforced policy to obtain the authorization decision and returns the decision to the application space. The application space then performs the operation on behalf of the client, if access to the operation is allowed.

Note that here we implemented `DynamicPolicyProvider` as the policy provider for our policy enforcement mechanism to dynamically modify the policy state. As the parameters of remote call to the `PolicyService` do not conform to Java's policy interface, nor is Java's policy interface available to remote clients, the policy engine service modifies the parameters and presents them to the `DynamicPolicyProvider`. Note that the context information is provided to the ADF by a file loaded at runtime and the policy rules are provided by an administrative graphical user interface (GUI). This GUI, known as `AccessControlGUI`, is described in Section 3.4.4.

The ADI sent by the client could have been sent in accordance with the AznAPI[1].

[1]http://www.opengroup.org/onlinepubs/009609199/toc.pdf

However, considering the fact that AznAPI is an API for C based access control framework and our prototype was built in Java, we chose not to use AznAPI for the purpose. We could have also made native calls to an underlying C implementation, but chose not to do so for simplicity.

3.4.2 Implementation

Figure 3.6 shows our implementation model for the remote authorization. We developed the remote authorization model to help us develop components that would be loosely coupled with each other. This would provide a flexible authorization scheme that can be configured as either a remote or a local authorization. One of the other considerations behind implementing this model was to make the code change required for remote authorization scheme to be backwards compatible with the local enforcement mechanism that we had developed as explained in the Section 3.3.

As shown in the figure, privileged resources are configured by the `AccessManager` component to use `PermissionChecker` for communicating with the `PolicyService`. `PolicyService` itself utilizes the `AccessManager` to enforce the dynamic role-based access control as discussed in Section 3.3.4. The policy authorization statements are enforced by the `PolicyHandler` components. The policy administration component puts the positive or negative authorizations to be signed by the `Policy Authority` into a space between the authority and the Policy Decision Point (PDP). These authorizations are then signed by the policy authority and put back to this same shared space for enforcement. This mechanism serves the following purpose:

- Allows only legitimate entries to be enforced by the `PolicyService`.
- Allows the shared space to serve as the central repository for policy tuples.
- Provides integrity by enforcing only policy tuples signed by the `Policy Authority`.

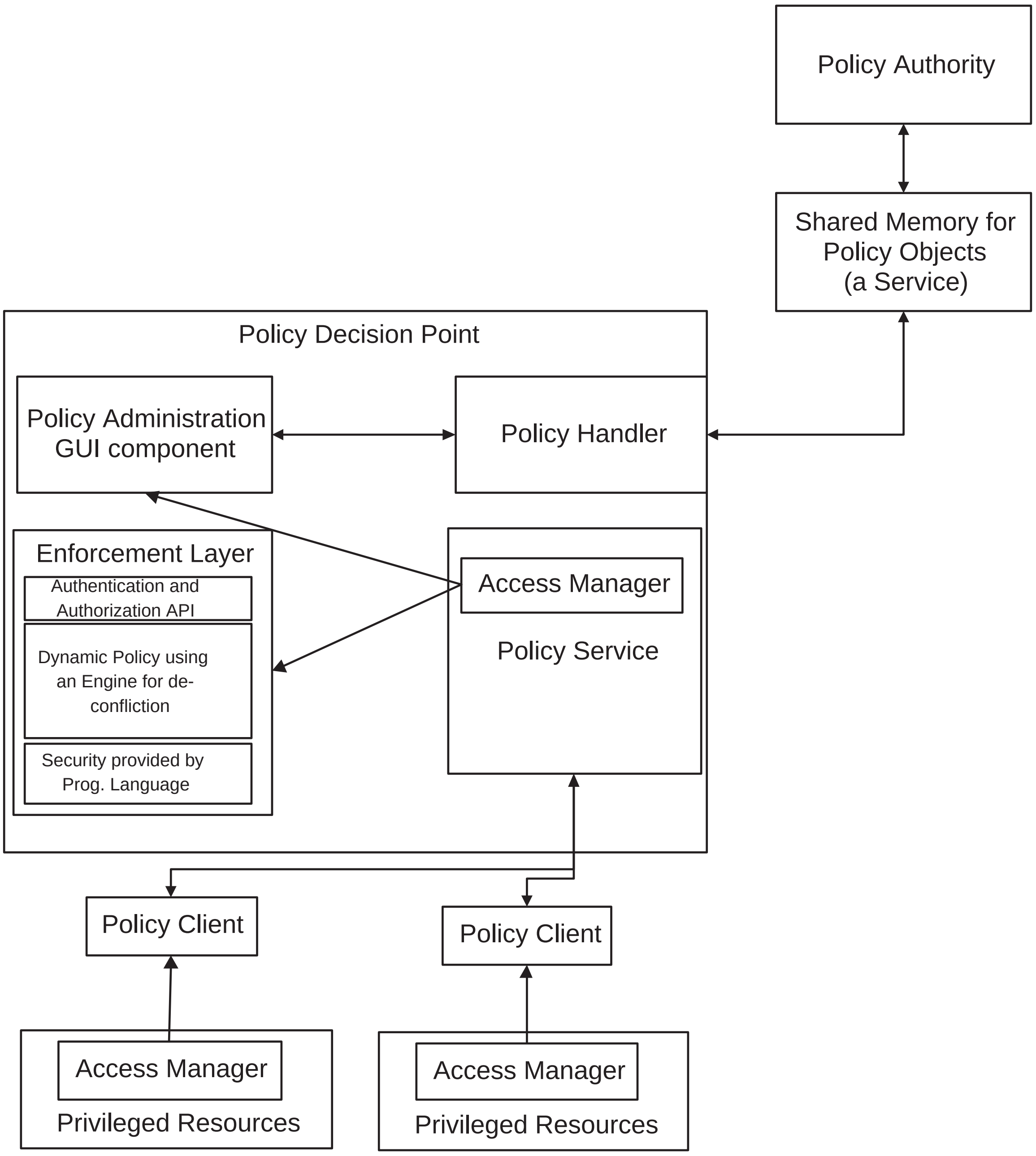

Policy Authority
Shared Memory for Policy Objects (a Service)
Policy Decision Point
Policy Administration GUI component
Policy Handler
Enforcement Layer
Authentication and Authorization API
Dynamic Policy using an Engine for de-confliction
Security provided by Prog. Language
Access Manager
Policy Service
Policy Client
Policy Client
Access Manager
Privileged Resources
Access Manager
Privileged Resources

Figure 3.6: Remote authorization architecture

We implemented the `PermissionChecker` components to be able to interact with `PolicyService`. `PolicyService` and privileged resources both utilize the `AccessManager` component. `AccessManager` is the only class that needs to be configured for (and aware of) where the authorization mechanism is present. The `PolicyService` is a Jini service that utilizes the `AccessManager` class, which takes care of enforcing security policy. This is done by utilizing the `DynamicPolicyProvider` class along with JAAS implementation for user-centric access control as explained in Section 3.3. This model lets us migrate from local policy enforcement at privileged resources to the remote authorization scheme with no code-level changes to the negative authorization scheme policy enforcement points.

3.4.3 Architecture Scalability

In accordance with the other components in Yalta, we required the policy enforcement mechanism to be scalable and survivable. Figures 3.7 and 3.8 show the scalability of the mechanism. Here, scalability is achieved with respect to two dimensions: depth and width.

Figure 3.7 shows how the remote authorization architecture is scalable with respect to depth of the system, the number of `PolicyService` components in the system. One or more intermediate `PolicyService` components are configured to contact higher level `PolicyService` components for their authorization decisions. This serves as delegation of the authorization mechanism. The `AccessManager` component can be configured at runtime by the `PolicyService` component to configure whether the policy will be enforced locally or authorization decisions should be obtained by contacting another `PolicyService` component located higher in the hierarchy.

Figure 3.8 shows how the remote authorization architecture is scalable with respect to width of the system, the number of domain components in the system. The multi-domain architecture here implies the various domains of authority that might be present in an environment such as a dynamic coalition. For example, consider a

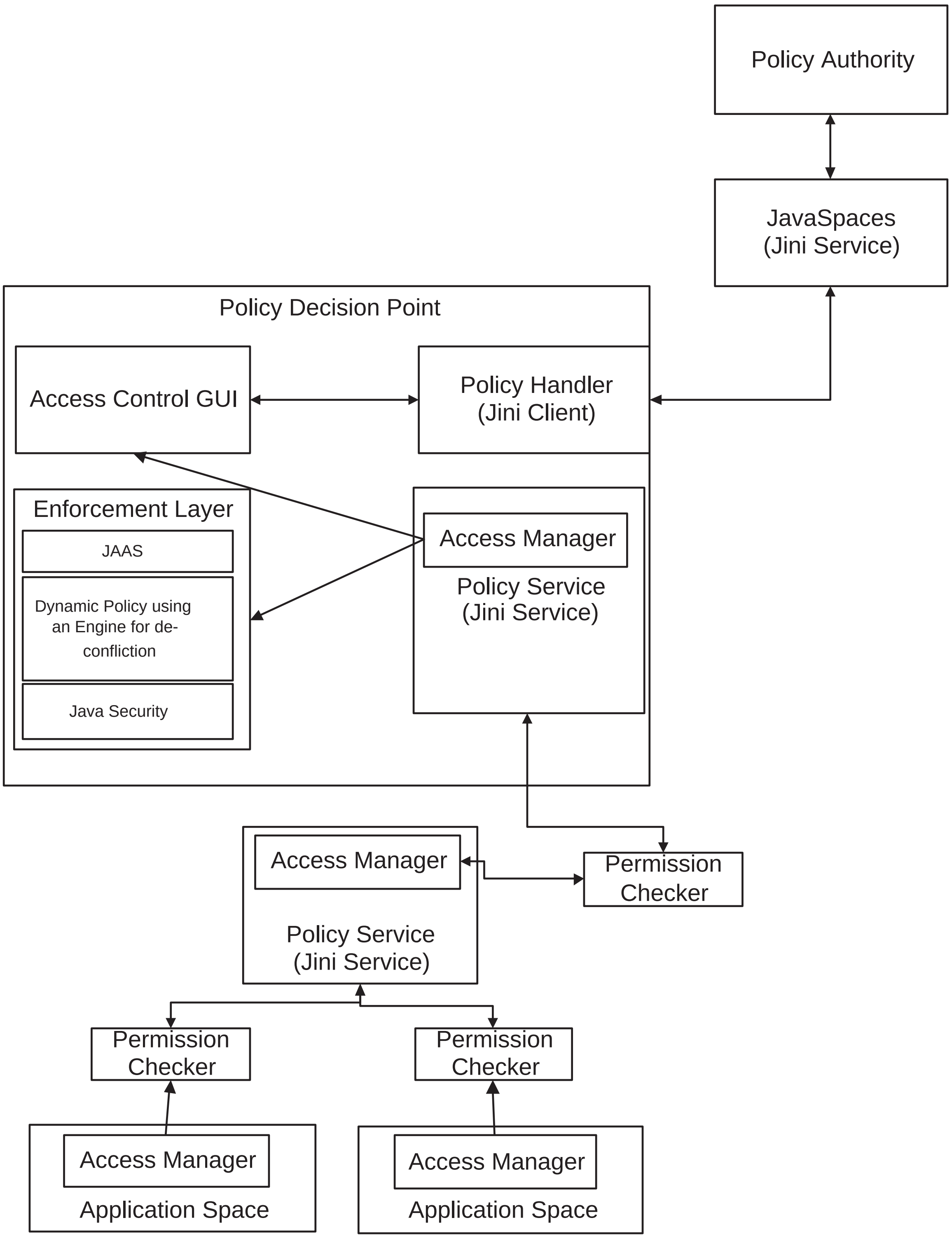


Figure 3.7: Scalable authorization: Hierarchy support

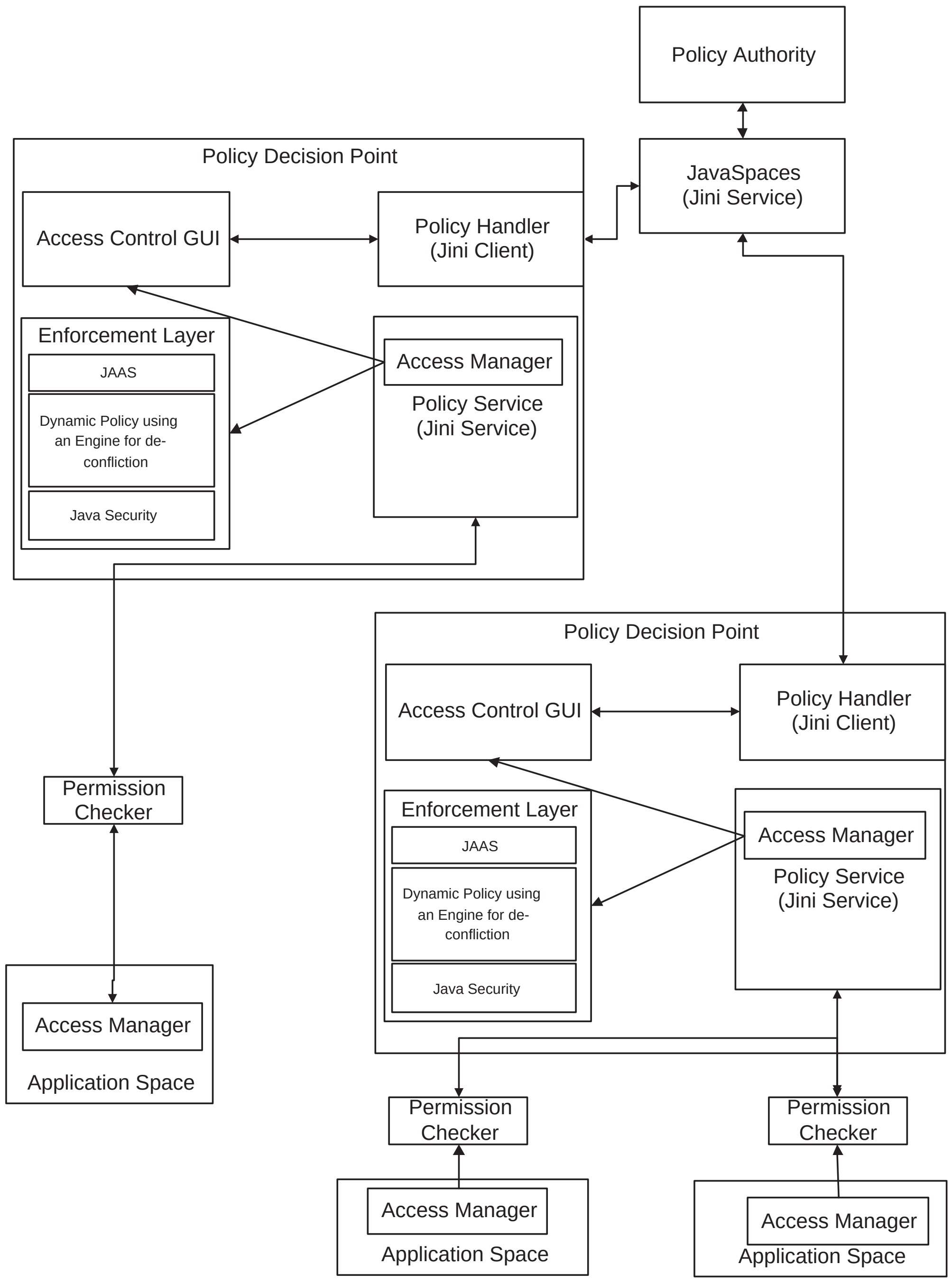


Figure 3.8: Scalable authorization: Multi-domain support

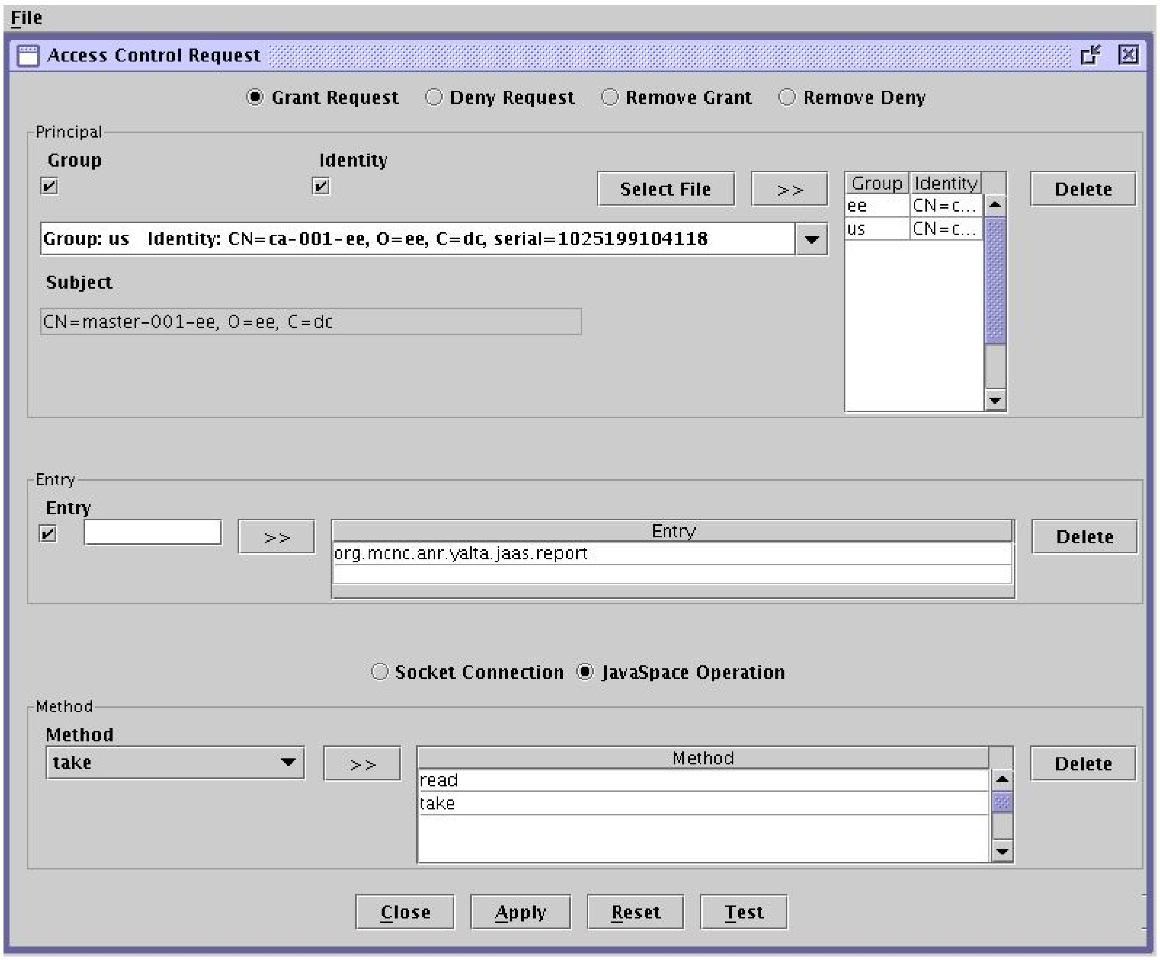
dynamic coalition of three countries that are allies during a war, implying three domains, in the simplest case: one domain per country. Each country would like to implement authorization mechanism for resources in their domains.

One or more `PermissionChecker` components can be configured to contact the `PolicyService` component from their domains for authorization decisions. In addition, many such `PolicyService` components can be registered with the shared policy space. These `PolicyService` components may or may not be from the same domain of authority. If there were multiple `PolicyService` components from the same domain, they would increase the fault tolerance in the system as downtime of any such component would mean that clients could contact another `PolicyService` in the same domain. If there were `PolicyService` components from the different domains, they would make the system scalable to include multiple domains. Such a scalable multiple-domain architecture is a core requirement of coalition environments [35].

Multiple `PolicyService` components can register with the higher level `PolicyService` components, and multiple higher `PolicyService` components can register to receive notification from the shared policy space to create a very powerful policy distribution and enforcement architecture required by mission critical applications.

3.4.4 Policy Administration

Policy administration takes place at the `AccessManager` service and has the benefit of not having to pass the policies to all the resources. This administration is handled by a GUI that presents a graphical interface to put policy signing requests into the shared policy space and obtaining the signed policies from the space on receipt of a notification from the space. The `AccessManager` interacts with the `DynamicPolicyProvider` class to ensure that these signed policies are enforced by the `Policy Decision Point`. The `AccessManager` has a component that listens to access check requests containing ADI sent by the `PermissionCheckers` from the ap-



Access Control Request

[✓ Grant Request] Deny Request Remove Grant Remove Deny

Principal

Group ✓ Identity ✓

Group: us Identity: CN=ca-001-ee, O=ee, C=dc, serial=1025199104118

Subject

CN=master-001-ee, O=ee, C=dc

Group	Identity
ee	CN=c...
us	CN=c...

Entry

Entry ✓

Entry
org.mcnc.anr.yalta.jaas.report

Socket Connection [✓ JavaSpace Operation]

Method

Method: take

Method
read
take

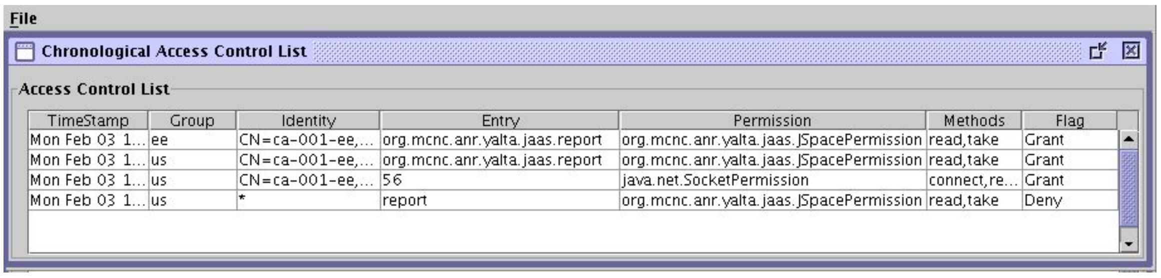
Figure 3.9: Policy administration interface

plication space and responds after executing the request on the user's behalf. All the interactions are done via RMI over SSL so that confidentiality and integrity is maintained. `PolicyService` is a Jini service and hence application spaces can discover the service and request access checks. Multiple domains are possible by specifying the domain in the startup script, which makes the authorization mechanism scalable.

Policy rules are obtained from the PKI space, which is where the `Policy Authority` of the distributed system publishes the signed policies. Since this is a JavaSpace, the `PolicyHandler` component started by the `AccessManager` registers with the JavaSpace to obtain notifications on new policy statements written into the space. The `PolicyHandler` fetches the policy whenever it receives a notification about a new policy entry for its domain in the shared space. This notification service is a standard Jini service and is utilized so that for each authorization call, the Policy Decision Point (PDP) does not have to seek for any updates into the space regarding the policy. The remote procedure call between the client and the `PolicyService` and between the `PolicyHandler` and space is by RMI over SSL to provide confidentiality and integrity.

Figure 3.9 shows the administrative component of this remote authorization scheme. We extract two attributes from the identity certificate of the requester: `GroupPrincipal` and `IdentityPrincipal`. The GUI enables one to grant, deny, revoke-grant, and revoke-deny access to subjects based on their principals. Two types of actions are currently included in the GUI: the JavaSpace operation and the Socket connection action. JavaSpace operation resembles any operation that a user intends to make on one or more entries in JavaSpace. Socket connection action represents the connection of the user to the JavaSpace. Socket connection check is sort of a first wall of defense against illegitimate users, while the JavaSpace action is more specific to a particular object, known as an `Entry`, or group of such objects.

This administrative GUI also allows the administrator to see the set of permissions applied by the GUI sorted in chronological order. Figure 3.10 is a snapshot of the access control elements applied by an administrator and is useful in policy logging purposes. This same GUI is used for administration in the case of local enforcement framework. The configuration details are managed by the `AccessManager` component and thus enables component re-use.



TimeStamp	Group	Identity	Entry	Permission	Methods	Flag
Mon Feb 03 1...	ee	CN=ca-001-ee,...	org.mcnc.anr.yalta.jaas.report	org.mcnc.anr.yalta.jaas.JSpacePermission	read,take	Grant
Mon Feb 03 1...	us	CN=ca-001-ee,...	org.mcnc.anr.yalta.jaas.report	org.mcnc.anr.yalta.jaas.JSpacePermission	read,take	Grant
Mon Feb 03 1...	us	CN=ca-001-ee,...	56	java.net.SocketPermission	connect,re...	Grant
Mon Feb 03 1...	us	*	report	org.mcnc.anr.yalta.jaas.JSpacePermission	read,take	Deny

Figure 3.10: Policy logging

3.5 Implementation Analysis

3.5.1 Merits

We have built a backwards compatible solution to utilizing negative authorizations with minimal changes required in the security classes of Java language.

A policy administrator can set up threshold levels to serve as warning mechanisms when the access check de-confliction value between a grant authorization and a deny authorization, described in Section 3.3.2, is numerically close. This would serve as an aid to the audit trail process where the authorities determine what authorizations were granted and denied by the system. Such a threshold level warning mechanism would aid the auditor to determine any holes in the policy enforced by the system.

With minor modifications, our implementation can also cater to the separation of duty principle at the administrative interface, in the absence of a constraints mechanisms in the language itself. For example, Java does not provide constraints in its access control framework. Instead of hard-coding the separation of duty policy for permissions in the application code, we can develop an input to the administrator interface for conflicting permissions. When a positive authorization is added to a particular role, the corresponding negative authorization(s) for separation of duty can be automatically issued by the system and vice-versa.

Remote authorization, along with negative permissions, makes the authorization process in the RBAC system scalable. Multi-domain RBAC systems can be realized using our approach, so it scales well to large-scale environments such as dynamic coalitions. Our remote authorization implementation is flexible for authorities to determine whether they want to use remote authorization or local authorization at startup instead of being bound by the remote authorization mechanism or local enforcement only.

We also support plugging in permission classes into the Java runtime to utilize this remote framework for multiple permission classes. This would be useful in cases of many different types of authorizations to be supported by the system.

3.5.2 Limitations

Caution should be observed while utilizing negative authorizations. A slight overlook in policy administration might lead to unnecessary denial of service to some authorized users. For example, an administrator adds a negative permission to a privileged resource for a junior role in the hierarchy and then adds a senior role that inherits all privileges from this junior role. An overlook on the part of the administrator might cause an unnecessary denial or service for a user in the senior role, if the senior role was supposed to have the access rights to that privileged resource. Consider the example given in Section 3.2. If one more role is added which is similar to a contract professional role, then it might lead to a violation of confidentiality of the system if the administrator fails to add denial privileges to the new role. Thus, negative authorizations should be carefully used and implemented ideally to cater to exception cases only.

The above two limitations are inherent due to the negative authorizations model and are not limitations of our implementation. We derive the role information for users through a monolithic architecture [34] where role information is embedded in the identity certificate of the user. This tight coupling between users and their roles was acceptable for our implementation, as our focus was implementing the denial mechanism and remote authorization to demonstrate scalability of our approach.

Chapter 4

Related Work

Role-based access control has been studied extensively and there are several excellent overviews of this methodology [38, 15].

One of the first solutions for utilizing negative rights in access control systems uses negative rights to signify explicit denial of access [42]. This system uses several de-confliction rules to reach the conclusion regarding the stronger authorization. For objects, inheritance relationships in the system is utilized for deriving access rights. For permissions, *include* and *imply* relationships are utilized for deciding whether the permission matches. An *include* relationship is the one that links a permission to another because of a containment relationship between permissions. For example, the rights of a particular workflow application include the rights of all the individual work required for that workflow. The *imply* relationship is used when a right for a stronger operation implies the right for a weaker operation. For example, a right to write to a file implies a right to append to that file. De-conflictions between *include* and *imply* relationships are resolved by giving preference to *imply* relationships as they are more tightly related to the system semantics, while the *include* relationship is just a grouping of rights used for ease of specification.

For subjects, the *take* and *have* relationships are utilized for deciding whether the permission matches. A *take* relationship is similar to the inheritance principle applied to objects. A *have* relationship is the one in which the user has some specific privileges of roles that he is performing. In case of conflicts, the *take* relationship is used first as *have* contains the rights that the user obtains indirectly from other users or roles that he may or may not take. Priorities for de-confliction rules are determined based on whatever is more efficient for the model. For example, a system implementing the access control using access control lists would use de-confliction based on object dimension. Our approach is a generic implementation of the subject dimension in their model. We have implemented the mechanism in a more transparent manner to applications, as our approach is based on the underlying security mechanism utilized in a widely used language. Any applications can make use of our mechanism without changes to their code.

Negative and positive authorizations via the use of strong and weak authorizations concept has also be studied [6]. This research caters specifically to database authorizations, though it can be applied to other contexts with minimal changes. The classification of weak and strong authorization helps de-confliction of the authorization rule. Any strong authorization over-rules the weak authorization, and only one strong authorization is allowed for a particular access permission. We do not utilize this concept of strong and weak authorizations as these can be sometimes overwhelming for the administrator to specify correctly, considering the fact that these strong and weak authorizations may be specified at any level in the role hierarchy.

The Andrew File System (AFS) [39] and Microsoft SQL Server 2000 [40] have negative permissions but those are simplistic cases where group and user identities are present. These systems have a model that implements the group based access control with negative permissions with no support for hierarchies in groups and the denial statements override grant statements. We permit the denial concept to hierarchical roles with an algorithm to de-conflict the negative and positive permissions.

Apache HTTP server[1] allows a more flexible mechanism by utilizing an `Order`

[1]http://httpd.apache.org

directive that specifies the order in which `Allow`, and `Deny` directives apply [11]. Accordingly one can place the directive value as:

- `DENY, ALLOW:` This represents an open policy where any client that does not match any deny directive *or* matches an allow directive is granted access.
- `ALLOW, DENY:` This represents a closed policy where any client that does not match any allow directive *or* matches a deny directive is denied access.
- `Mutual-failure:` This represents a restrictive policy in which only clients that do not match any deny directive and match an allow directive are granted access.

These options signify either a highly closed or a highly open policy, whereas our mechanism supports more fine-grained control over the authorization, depending on the hierarchy level at which the grant/deny authorizations are present.

Coalition environments, which require collaborative effort on the part of participating organizations, often require the system to enforce access control decisions on users based on teams that the user participates in. For such environments, other higher level models such as Team-based access control (TMAC) [49] are proposed. Such environments need a flexible model that includes role-based permissions for objects, yet requires finer controls such as identity-based controls on individual users in certain roles and individual object instances. One such example for fine-graining is a clinical policy stating that no clinical staff should be allowed to access a patient's records unless they were providing care for that patient [49]. Such a requirement can be met by the RBAC model, although it is cumbersome at best. The negative authorizations model would also not pose any significant advantage or disadvantage for such a requirement.

Access control constraints [2] are a means to providing flexibility in the system for catering to *exception* cases. However, these constraints come at some implementation complexity. For resources not requiring a high level of sophistication, these constraints can be quite overwhelming; they increase the complexity of the parser-generator required to read and modify the access control elements or rows that govern access to a resource. Consider the case of exceptions to role based access privilege where a user is allowed to access a resource not based on a role but on his authority in the organization. Or consider that a user is not allowed to access a particular resource in spite of his being in an authorized role for that resource due to his earlier abuse of that resource. These can be either expressed by constraints or by negative authorizations. For systems requiring a simple mechanism to manage access control system catering to these exception cases, an RBAC system with negative authorizations should be an effective means of administration. These negative authorizations can also complement constraints in more sophisticated systems. Constraints generally act as pre-conditions to the access control rules, while negative authorizations are actually implemented as tuples or elements of the access control policy.

Akenti [26] is an authorization infrastructure for widely distributed resources and utilizes X.509 attribute certificates for access control. Akenti's gateway (AEF) authenticates via the identity certificate of clients and passes the identity information in the access request sent to a Policy Engine (ADF) for evaluation. The ADF extracts the attribute certificates from the public directories and evaluates them against the use-conditions given by different domain of authorities to derive at an access decision. Due to the search time for certificates and their validation, this process might be time-constrained. We enforce the use-condition or policy statements *a priori* at our `PolicyService` component. In our implementation, the AEF extracts the role information from the certificate presented by the clients and passes them to the ADF. Therefore, the access control decision by the ADF is not time-constrained. A client might cause many access control requests, one for each resource it wants, to their ADF component. Thus, it may be beneficial in many cases to extract the attribute information from the certificates at the AEF rather than the ADF in some cases. For such cases, our implementation might prove to be beneficial. Our mechanism also provides the benefit that the client can activate whatever attribute he wants, rather than having a publicly availably directory of attribute certificates. Denial-of-service attacks due to non-availability of these directories is a major threat in the Akenti approach. However, a downside to our approach might be the requirement to implement a revocation scheme for certificates. This revocation scheme, for attribute certificates, is not required for the Akenti approach.

The Secure Virtual Enclaves (SVE) [41] project has concentrated on many policy issues, such as accomodating different role hierarchies, multiple dynamic coalitions, and type enforcement, which were beyond the scope of our work. SVE has an authorization model similar to ours. However, the scalability issue for authorization model was not addressed by the project. We have focussed on scalability of our implementation and have provided a scalable and flexible authorization mechanism that can be implemented at the discretion of the domain authorities. Our mechanism can be easily extended to achieve scalability with respect to depth of a domain hierarchy via use of multi-tier client-server access resolution.

SVE does not support policies that are are applicable across all domains. In addition to the local domain policies, many coalitions would require an infrastructure that supports such cross-domain policies. We provide both domain-wide and cross-domain policies via a hierarchical approach containing shared coalition authority [43] above domains. This shared coalition authority performs the task of reviewing and granting such domain-wide and cross-domain policies. SVE lacks an infrastructure to support cross-domain policies efficiently.

Access Manager [24] is a product that implements a remote authorization model similar to ours. They also utilize the JAAS framework and extend it to include remote authorizations. This remote authorization framework consists of the Resource Manager, Authorization Server and the Policy Server. A Resource Manager makes calls to a pre-configured Authorization Server that takes the access control decision based on the policy present at the Policy Server. The Access Manager API provides a permission class that establishes an SSL-protected socket in its static initializer to talk to Access Manager.

Our implementation has several unique features like negative authorizations, scalable framework and multi-domain infrastructure. Our implementation is very flexible, in the sense that Policy Enforcement Points can determine whether they want to use remote authorization or local authorization at startup instead of being bound by the remote authorization mechanism or local enforcement only. We also support plugging several `Permission` classes into the Java runtime environment by extension of our `PrivilegedPermission`. Such a mechanism allows flexibility as customized authorization classes can be built to utilize remote authorization.

Chapter 5

Conclusion

5.1 Contributions

This thesis proposed utilizing negative authorizations to cater to exceptions in access control and provided a case for including negative authorizations in the RBAC model. We showed a reference implementation of an RBAC system utilizing negative authorizations in a widely accepted programming language with minimal changes and requiring no changes to existing application code. Because we have implemented our negative authorizations scheme into a widely used programming language, and in an application transparent manner, our scheme can be utilized by all resources wanting to utilize dynamic administration of security policies.

We demonstrated the design and implementation of a scalable RBAC implementation that can be used in dynamic large scale environments. We showed the scalability of our implementation with respect to the depth of the environment and the support for multiple domains in the environment. We developed a flexible architecture to configure the authorization scheme depending on the requirements of such an environment. Our remote authorization scheme has been implemented as a plug-in for an existing application independent authorization framework (JAAS) and can be utilized by all resources using the JAAS framework for their authorization. We hope our remote authorization implementation leads to more research in terms of scalability of the RBAC environment and our abstraction model helps guide towards more scalable architectures.

We successfully demonstrated the implementation of the negative authorizations scheme and scalability of remote authorization architecture at the third DARPA Information Survivability Conference and Exposition (DISCEX III) [43].

5.2 Future Work

We have currently implemented only the RBAC1 level of the RBAC model. We utilize the SSL handshake as the session activation mechanism and bind role information within the X.509 identity certificates. User-role assignment is done after the user X.509 certificate is validated, when we extract the role information from the certificate. Role hierarchies are supported by specifying the hierarchy at runtime via a static file. We intend to investigate more about complementing negative authorizations with constraints, and supporting a more flexible mechanism to derive role information after user authentication.

We have implemented negative rights and de-conflictions for only one dimension of the access control model: subjects. We expect similar work be done in areas of inheritance models for permission rights and objects. However, another important question to be answered would be in what order should the different dimensions of the access control model be considered [42].

As discussed in Section 3.4.3, our system can be configured to delegate authorization decisions to higher levels in the hierarchy or take authorization decisions locally. This mechanism can be extended to selectively delegate authorization decisions depending on various parameters, such as sensitivity or granularity of the permission. This would require a more sophisticated `AccessManager` component that can determine which authorizations have to be delegated and which have to be enforced locally. This would also require a sophisticated policy distribution hierarchy that can selectively distribute the required policy tuples into `PolicyHandler` for enforcement.

The `DynamicPolicyProvider` component can be extended to dynamically include changes in the role hierarchy. This might be a requirement for many dynamic systems where the role hierarchies might change. The current algorithm for de-confliction between negative and positive authorizations is less sophisticated where all the applicable positive and negative authorizations are checked to derive the priority value of the authorization. This mechanism can be extended to include more sophisticated algorithms such as keeping the authorizations lists sorted and alternate the checking of positive and negative list tuples, or indexing these authorization lists as per the hierarchy of roles for more efficient lookup and results.

The existing implementation can be extended in several ways to adhere to existing standards such as utilizing X.509 attribute certificates [13] for signed policy objects and utilizing SAML [21] language for the access control request-response communication.

We plan to study the use of negative certificates [22] for implementing negative authorizations and discuss how our implementation complements their model for assigning roles in open and distributed environments.

Bibliography

[1] ISO/IEC 10181-3: Security frameworks for open systems: Access control framework, 1996.

[2] Gail-Joon Ahn and Ravi Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and System Security (TISSEC)*, 3(4):207–226, November 2000.

[3] Yair Amir, Giuseppe Ateniese, Damian Hasse, Yongdae Kim, Cristina Nita-Rotaru, Theo Schlossnagle, John Schultz, Jonathan Stanton, and Gene Tsudik. Secure group communication in asynchronous networks with failures: Integration and experiments. In *The* 20^{th} *International Conference on Distributed Computing Systems (ICDCS)*, April 2000.

[4] Anne Anderson. Java^{TM} access control mechanisms. Technical Report TR-2002-108, Sun Microsystems, March 2002.

[5] Elliot D. Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE, March 1973.

[6] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. A flexible authorization mechanism for relational data management systems. *ACM Transactions on Information Systems (TOIS)*, 17(2):101–140, April 1999.

[7] Reinhardt A. Botha and Jan H.P. Eloff. Designing role hierarchies for access con-

trol in workflow systems. In 25th *Annual International Computer Software and Applications Conference (COMPSAC)*, pages 117–122. IEEE Computer Society Press, October 2001.

[8] David F.C. Brewer and Michael J. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society Press, May 1989.

[9] Gregory T. Byrd, Fengmin Gong, Chandramouli Sargor, and Timothy J. Smith. Yalta: A secure collaborative space for dynamic coalitions. In *IEEE Workshop on Information Assurance and Security*, pages 30–37. IEEE Computer Society Press, 2001.

[10] Eve Cohen, Roshan K. Thomas, William Winsborough, and Deborah Shands. Models for coalition-based access control (CBAC). In 7th *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 97–106. ACM Press, June 2002.

[11] Sabrina De Capitani di Vimerati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. In *Software – Practice and Experience*, volume 33, pages 397–421, April 2003.

[12] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, January 1999.

[13] S. Farrell and R. Housley. RFC 3281: An internet attribute certificate profile for authorization, April 2002.

[14] David F. Ferraiolo and Richard Kuhn. Role-based access controls. In 15th *NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[15] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control.

ACM Transactions on Information and System Security (TISSEC), 4(3):224–274, August 2001.

[16] Eric Freudenthal, Tracy Pesin, Lawrence Port, Edward Keenan, and Vijay Karamcheti. dRBAC: Distributed role-based access control for dynamic coalition environments. Technical Report TR2001-819, New York University, 2001.

[17] Luigi Giuri. Role-based access control in Java. In *Proceedings of the* 3rd *ACM Workshop on Role-based access control*, pages 91–100. ACM Press, October 1998.

[18] Virgil D. Gligor, Serban I. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *IEEE Symposium on Security and Privacy*, pages 172–183. IEEE Computer Society Press, May 1998.

[19] Li Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* AddisonWesley, June 1999.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition.* Sun Microsystems, April 2000.

[21] Phillip Hallam-Baker and Eve Maler. Assertions and protocol for the OASIS Security Assertion Markup Language (SAML), November 2002.

[22] Amir Herzberg, Yosi Mass, Joris Mihaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, pages 2–14. IEEE Computer Society Press, May 2000.

[23] Thomas Hildmann and Jrg Barholdt. Managing trust between collaborating companies using outsourced role based access control. In *Proceedings of the* 4th *ACM Workshop on Role-based access control*, pages 105–111. ACM Press, October 1999.

[24] IBM Corporation. *IBM Tivoli Access Manager: Authorization Java Classes Developers Reference*, April 2002.

[25] Trent Jaeger and Atul Prakash. Requirements of role-based access control for collaborative systems. In *Proceedings of the* 1st *ACM Workshop on Role-based access control*, page 16. ACM Press, 1996.

[26] William Johnston, Srilekha Mudumbai, and Mary Thompson. Authorization and attribute certificates for widely distributed access control. In 7th *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET'ICE)*, pages 340–345. IEEE Computer Society Press, June 1998.

[27] Himanshu Khurana, Virgil Gligor, and John Linn. Reasoning about joint administration of access policies for coalition resources. In 22nd *International Conference on Distributed Computing Systems (ICDCS)*, pages 429–438. IEEE Computer Society Press, July 2002.

[28] Himanshu Khurana and Virgil D. Gligor. Review and revocation of access privileges distributed with PKI certificates. In *Security Protocols, Lecture Notes in Computer Science (LNCS)*, volume 2133, pages 100–112. Springer-Verlag, September 2001.

[29] John Kohl and Clifford Neuman. RFC 1510: The Kerberos Network Authentication Service V5, September 1993.

[30] John McLean. Security models. In *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[31] SangYeob Na and SuhHyun Cheon. Role delegation in role-based access control. In *Proceedings of the* 5th *ACM Workshop on Role-based access control*, pages 39–44. ACM Press, July 2000.

[32] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(2):85–106, May 2000.

[33] Jaehong Park and Ravi Sandhu. Towards usage control models: Beyond traditional access control. In 7th *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 57–64. ACM Press, June 2002.

[34] Joon S. Park and Ravi S. Sandhu. Binding identities and attributes using digitally signed certificates. In 16th *Annual Computer Security Applications Conference (ACSAC'00)*, pages 120–127. IEEE Computer Society Press, December 2000.

[35] Charles E. Phillips, Jr., T.C. Ting, and Steven A. Demurjian. Information sharing and security in dynamic coalitions. In 7th *ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 87–96. ACM Press, June 2002.

[36] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, 1975.

[37] Ravi Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[38] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.

[39] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, 1990.

[40] Saskia Schott, Jerry Spivey, Jim Skinner, Cathan Cook, and Allan Hirt. *Microsoft SQL Server 2000 Operations Guide*. Microsoft Press, July 2002.

[41] Deborah Shands, Richard Yee, Jay Jacobs, and E. John Sebes. Secure virtual enclaves: Supporting coalition use of distributed application technologies. *ACM Transactions on Information and System Security (TISSEC)*, 4(2):103–133, May 2001.

[42] HongHai Shen and Prasun Dewan. Access control for collaborative environments. In *ACM Conference on Computer-Supported Cooperative Work (CSCW)*, pages 51–58. ACM Press, November 1992.

[43] Timothy J. Smith, Gregory T. Byrd, Xiaoyong Wu, Hongjie Xin, Krithiga Thangavelu, Rong Wang, and Arpan Shah. Dynamic PKI and secure tuplespaces for distributed coalitions. In 3^{rd} *DARPA Information Survivability Conference and Exposition (DISCEX III)*, April 2003.

[44] Sun Microsystems. *Java Remote Method Invocation Specification Rev. 1.7*, December 1999.

[45] Sun Microsystems. *JavaSpaces Specification Version 1.0*, 1999.

[46] Sun Microsystems. *Jini Architecture Specification Version 1.1*, October 2000.

[47] Sun Microsystems. *Java^{TM} 2 SDK, Standard Edition Documentation*, 2002.

[48] Sun Microsystems. *Davis Early Access Draft Specifications Release*, February 2003.

[49] Roshan K. Thomas. Team-based access control (TMAC): A primitive for applying role-based access controls in collaborative environments. In *Proceedings of the 2^{nd} ACM Workshop on Role-based access control*, pages 13–19. ACM Press, November 1997.